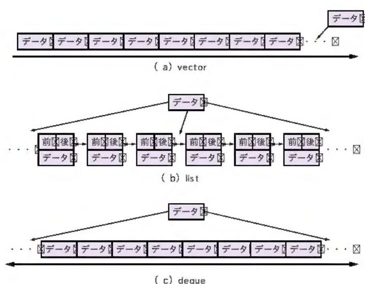
Interface

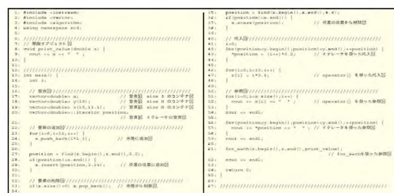
2004
2

Contents



[表紙デザイン：㈱プランニング・ロケッツ]





Interface

2004
2
Contents

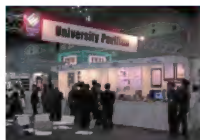
連載「開発技術者のためのアセンブラ入門」，「開発環境探訪」，「『VxWORKS』を使ったRTOS技術の基礎と応用」は，お休みさせていただきます．

▶ 組み込み技術の総合展示会

Embedded Technology 2003

北村 俊之

「デジタル時代の最先端組込み技術を一堂に」をテーマに「ESEC」と並ぶ組み込み技術の総合展示会「Embedded Technology 2003」(旧称：MST)が11月12日(水)～14日(金)の3日間，パシフィコ横浜で開催された．主催は(社)日本システムハウス協会．今回は，組み込みシステムのアプリケーションを意識し，「デジタルコンシューマ/マルチメディア」，「オートモーティブ/テレマティクス」，「モバイル/ユビキタス」の三つの応用テーマを掲げ，これらの分野で躍進する各出展社の最新システムの展示・実演を中心として紹介する構成となっていた．特別展示として「インドパビリオン」をはじめとした海外からの出展や，大学研究室の出展コーナーである「ユニバーシティパビリオン」(写真1)も充実が図られていた．さらに「ITSパビリオン」，「ベンチャービレッジ」など時代のニーズに応えた出展企画も多く見られた．



〔写真1〕ユニバーシティパビリオン

また，展示会の2日前から，100プログラムを超えるカンファレンスが開催され，こちらも昨年に引き続き大盛況であった．最終的な延べ来場者数は3日間で19,201人に達した．

● 来場者の関心が高い「T-Engine」と「組み込みLinux」

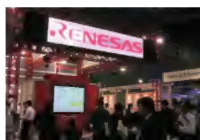
今年後半の組み込み技術関連のもっとも大きな話題としては，9月に発表された「T-Engine」と「Windows CE.NET」の提携があげられる．こうした状況を反映しているのか，今回は例年にも増してT-Engine関連製品の出展が多かった．

BTRON仕様OS「超漢字」などを開発，販売しているパーソナルメディアでは，SH3-DSPを搭載した「T-Engine/SH7727開発キット」をはじめ，SH-4，MIPS，ARM，M32Rなどの各種T-Engine開発キットを展示していた．また，同ブースではブース内に設置した四つの小型ディジタルカメラから送られてくる映像を受信するT-Engine端末のデモも行っていた(写真2)．

アームでは，ブースをプレゼンテーション，ARM展示，パートナー展示の三つのコーナーに分けて展示を行っていた．パートナー展示コーナーでは，今回も20社以上のパートナー企業が参加し，各社のツールやソフトウェア，ハードウェアなど広範囲なソリューション展示で来場者の高い関心を集めていた．こちらでも，T-Engineコーナーを設け，ARMアーキテクチャが内蔵されたT-Engineボードの展示を行った．


〔写真2〕T-Engine ARカメラ．四つの映像を同時受信している

ルネサス テクノロジ(写真3)のブースでも，同社のマイコンの開発環境「R8C/Tinyシリーズ」，「M16C/Tinyシリーズ」やSH-Linuxデモなどに交じって「SH7751R T-Engine USB2.0プロトコルスタックデモ」，「SH7760 T-Engine PMC T-Shellデモ」などが披露されていた．



〔写真3〕ルネサス テクノロジのブース

T-Engine同様，組み込み関連で最近，著しい躍進を遂げているのが「組み込みLinux」だろう．こちらもメトロワークス，モンタビスタソフトウェアジャパン，日新システムズ，コンピューテックスなど多くのブースで，開発環境から評価ボードまで多彩なソリューションが展示されていた．組み込みLinuxの普及にもっとも力を入れているディストリビュータの一つである，モンタビスタソフトウェアジャパンでは，同社の主力製品である「MontaVista Linux」を中心とした展示を行っていた(写真4)．今回も「Professional Edition」，「Consumer Electronics Edition」，「Carrier Grade Edition」など，各セグメントに合わせたラインナップで来場者の関心を集めていた．


〔写真4〕MontaVista Linuxを搭載した Motorola製携帯電話

コンシューマエレクトロニクスとワイヤレス開発ソリューションを今年度のメインテーマとしているというメトロワークスでは，開発サイクルのすべてのステージをサポートするトータルソリューションに力を入れたとのことだった．ブースでは，主力製品である「CodeWarrior」を中心とした，組み込みLinux，ホームサーバ，モバイルデバイスプラットホーム，ワイヤレス統合開発環境，2D/3Dグラフィックス関連などの展示をデモを交えながら行っていた．

コンピューテックスでは，組み込みLinuxに最適なアプリケーションデバッガ「CSIDE for Linux ARM，SuperH，PowerPC」やカーネル/ローダブルモジュールデバッガ「NEXTiCE/PALMiCE+Linux-DBGLIB」の展示を行っていた．「NEXTiCE/PALMiCE」は，組み込み機器の主流となるCPUのオンチップデバッグをサポートする，パームサイズのコンパクトなデバッガで，人気の高い製品だとのことだった．

● その他の注目製品

エナジーセービングをキーワードに次世代携帯機器の開発をサポートするエプソンでは，32ビットマイコン「S1C33ファミリ」(写真5)やUSB On-The-Goコントローラ，無接点電力モジュールなどの製品を展示していた．中でも携帯機器向けの漢字ROM内蔵マイコンは，世界でも初の試みだという．本製品は，漢字圏に向けた製品に対し，マイコン＋漢字ROM+LCDドライバを1チップ化したものである．これによりMDプレーヤなどの小型液晶ディスプレイに漢字を表示できるようになるという．展示会場ではJIS第一水準/第二水準に対応した表示デモを行っていた．


〔写真5〕S1C33ファミリを用いた音声認識/合成のデモ

日本ノーベルでは，組み込みシステム用コンパイラ/ITRON OSの品質評価からHMI部分の自動検証まで幅広いサポートを紹介していた．とくに，HMI自動評価システムに関しては，これまでの携帯電話の自動検証システムをさらに発展させ，民生機器全般の自動評価が行えるシステムを参考出品していた．これによって，高機能化する民生機器の検証が，飛躍的に向上するという．

ウインドリバーでは，TORNADO/VxWORKSに各種ミドルウェアを統合し，最新のCPUやIPv6，無線LANなどに対応した「WIND RIVER PLATFORM」およびマルチコア対応JTAG ICE「WIND POWER ICE/IDE」などの展示を行った．同製品では，一つのICEで複数のコア，CPUへの同時デバッグを実現しているという．また，同社の技術を搭載した民生機器も多数展示されていた(写真6)．

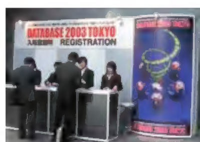

〔写真6〕VxWORKSを搭載した製品の数々．DiMAGE7，FinePix

▶ 商用データベースの総合展示会

DATABASE 2003 TOKYO

北村 俊之

「ほしい情報が必ず見つかる，情報共有・活用のノウハウが必ずわかる“情報マーケットプレイス”」をテーマに「DATABASE 2003 TOKYO」が10月29(水)～31(金)の3日間，東京国際フォーラムで開催された(写真1)．主催は(財)データベース振興センター(DPC)，日本データベース協会(DINA)．今年で第15回目を迎える本展示会は，業界の最新情報とトレンドを紹介する場になっている．



〔写真1〕会場入り口のようす

「DATABASE 2003」という名称からオラクルやSQLサーバ，サイベースといった，データベースシステム構築用ツールあるいはソリューションを期待して本展示会を訪れた方は，違和感を感じたかもしれない．というのも，これらのデータベースシステムベンダは，ほとんど出展しておらず，ユーザーが直接利用できる商用データベースなどの出展をメインに据えているからだ．

今回は例年にも増して，XMLによるドキュメントの管理，ネットにおけるドキュメントの検索性向上などのソリューションをアピールするブースが数多く見られた．ここ数年来，各方面で注目が集まっている，GIS(地理情報システム)関連のベンダが数多く出展していることも，本展示会のもう一つの見所だった．さらに，「産学連携/TLOコーナー」も設けられ，大学からの出展もいくつか見られた．主催者セミナーとして，G-XML国際セミナー「応用が進む，G-XML/GML」，同時開催セミナーとして，第8回SGML/XML研修フォーラム「XMLとデータベースを活用した新しいビジネスモデル」も開催され，こちらも盛況だった．

● 地図データベース& GIS

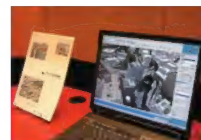
今回もこの分野では，多くのベンダやメーカーが出展しており，例年どおりの活況を呈していた．ゼンリンでは，地図データベース「Zmap-TOWN II」，「Zmap-AREA II」，GISソフトウェア「MAPiZM」，「Zmap-Office」，「OA-Light II」などを展示していた．「Zmap-TOWN II」は住宅地図をベクトル形式でデータベース化したものである．また，「MAPiZM」はエリアマーケティングなど幅広い分野に対応できるWebGISで，高度なエリア解析機能を装備していることが特徴だという．

WebGISエンジン「Maplet」と，次世代Webファイリングシステム「EFD」との融合ソリューションを展示していたのがコボプラン(写真2)である．このソリューションにより，位置/空間情報とリンクしたファシリティマネージメントやナレッジマネージメントを実現するという．「EFD」は，高速ビューイングやブラウザのみでのファイル閲覧機能など，ファイリングシステムに求められるほとんどの機能を実現しているため，電子納品やCALS/EC，ISOなどの各規格に対応したシステムの構築も可能である．


〔写真2〕コボプランのデモ

ワイ・ビー・シーでは，地図情報システム「MAP-STARシリーズ」の展示を行っていた．これは顧客情報と地図表示システムを連携させた統合システムで，用途に応じて「MAP-STAR SD」，「MAP-STAR LT II」，「Point-Star」などの種類がある．「Point-Star」は，ポイントカードシステムに「MAP-STAR」を搭載したシステムで，実績などの結果を地図上で分析できる．

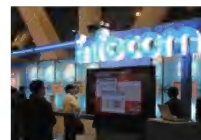
インクリメントPのブースで来場者の注目を集めていた展示が，「MAPCUBE」のデモ(写真3)である．同製品は，街の情景をリアルに表現する3Dマップデータである．また，同社では，カーナビゲーションで採用されているディジタル地図をGIS向けの背景ベクトルデータとしても提供している．


〔写真3〕「MAPCUBE」のデモ画面

● ドキュメント管理& XMLソリューション

従来は，ドキュメント管理というとSGMLが主流だったが，ここ数年急速に注目を集めているのがXMLだろう．こうした状況を反映してか，本展示会でもXMLを中心としたドキュメント管理ソリューションが数多く展示されていた．XML・RDBダイレクト連動のサーバサイドパブリッシングシステムを全面に打ち出していたのが，シンプルプロダクツである．同社のシステムは，製品カタログ，マニュアル，各種販促物などをXMLやRDBからダイレクトに自動レイアウトして出力できる．

インフォコム(写真4)では，企業ドキュメント管理システム「MyQuick」，製薬企業向けR&Dドキュメント管理システム「Pharma Potal/RD」，統合特許管理システム「TOPAN」など，企業情報のあり方に着目した多彩なソリューションを展示していた．

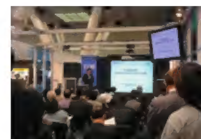


〔写真4〕インフォコムのブース

サピエンスでは，イントラネット対応業務用ドキュメント管理システム，ブロードバンド時代のWeb出版支援サービス「プラグインフリー・ウェブリッシング」の展示デモを行っていた．「プラグインフリー・ウェブリッシング」は，ビューワのインストールなしにペーパーイメージオンWebを実現するシステムである．紙ベースの出版物のオンラインビジネスを，コンテンツの入稿からWeb公開/運用までトータルにサポートするサービスである．

● サービス&パッケージ

例年，ワークショップとプレゼンテーションセミナーで，来場者の高い関心を集めているブースが帝国データバンクである(写真5)．今回は，独自の定性情報を元にした，企業が1年間に倒産する確率を提供する「倒産予測値」を中心とした，各種与信管理サービスの紹介していた．同社が長年蓄積してきた情報やノウハウをもとに，与信管理方法の一つとして提案している「信用リスク管理」への多面的なアプローチについてのプレゼンテーションも行っていた．


〔写真5〕来場者で賑わう帝国データバンクのブース

スマートスタイルでは，MySQL用データベース管理ソフト「MySQL Studio Navicat」(PremiumSoft社製，写真6)の展示デモを行っていた．同製品では，MySQLとWebサーバの効率的な活用ができ，ApacheまたはPHPとMySQLをWindows上で利用できるようになる．GUIを利用したデータベース，テーブル，ユーザー権限の作成/管理，視覚的なクエリの作成，リアルタイムなMySQLサーバの状態監視などを特徴としている．


〔写真6〕「MySQL Studio Navicat」の管理画面

2003国際ロボット展

■ 日時：2003年11月19日（水）～22日（土）
■ 場所：東京国際展示場（東京ビッグサイト，東京都江東区）

（社）日本ロボット工業会が2年に一度開催するロボットに関する展示会「国際ロボット展」．15回目の開催となる今回のテーマは，「RT（ロボットテクノロジ）が未来を拓く－モノづくりからパーソナルまで－」だった．

117社27団体が出展し，産業用ロボットを中心に，さまざまなロボットや，それらを支える関連機器や技術などが686のブースで展示された．

産業用ロボットのブースでは，自己判断機能をもつロボットも展示された．ファナックでは，レーザレンジファインダやカメラからの入力情報を用いてワーク（ロボットの作業対象物）を識別し，ワークの位置を判断して作業するいわゆる知的ロボットを発表した．

また，セイコーエプソンのブースでのμFRのデモは，人だかりができるほどの盛況ぶりだった．このデモは，重さ約8.9g，直径約130mm×高さ30mmの世界最小ロボットであるμFRを実際に動かすというものだった．このロボットは，二つの薄型の超音波モータによって，2枚のプロペラを回転させて浮上し，リニアアクチュエータによって重心を移動し，空中での姿勢を制御する（ただし，ノイズには弱いらしく，カメラでのフラッシュ撮影は禁止）．電源はケーブルを通じて外部から供給する．

そのほか，さまざまなエンターテイメントロボットや各研究機関が開発したロボットなどが展示された．



セイコーエプソンのμFRのデモ



箱（ワーク）の積み上げや積み下ろしを行うファナックのロボット．近くにあるワークはカメラに大きく写ることから距離を，箱に印刷してある文字から箱の傾きぐあいを判断する



受付のようす．平日（金曜日）にこれだけの人が集まった


ビジネスデザイン研究所のコミュニケーションロボット「ifbot」



産業総合研究所のヒューマノイドロボット「HRP-2」も登場．床に寝転んで起き上がるなどのデモも披露した

第2回日本イノベーター大賞表彰式

■ 日時：2003年11月28日（金）
■ 場所：新高輪プリンスホテル（東京都港区）

日経BP社は，日本イノベーター大賞として，トロンOSを開発した坂村健氏と，カーボンナノチューブを発明したNEC特別主席研究員の飯島澄男氏の2名を，優秀賞としてインクスの社長である山田眞次郎氏を表彰した．同賞は，日本の産業界において，独創的な技術やアイディアで活躍する日本人を選ぶものである．


トロンOSの開発者である坂村氏（中央）


カーボンナノチューブを発明した飯島氏（右）


携帯電話の試作機などの金型メーカーであるインクス社長の山田氏（右）

ハッカーの常識的見聞録



今月の常識

Longhorn を体感しよう!

広畑 由紀夫

☆日本において，12月9日に開催される「.NET Developer Conference」に先立ち，MSDN会員の開発者向けの配布が始まりました．早速，簡単にレビューしてみます．

今回配布されたLonghornは，β版よりさらに前のPRE-ALPHA版で，基本的に動作確認や検証を行うためのものではなく，内部APIや新機能の一部を試してみるためのものです．そのため，PRE-ALPHA版で今後のLonghornを語るのは無謀なことなのですが，今後の方向性をみるという面では，今回の開発者向けの配布開始は大きな意味があるといえるでしょう．

● Virtual PC 2004

今回配布されたLonghornには，動作確認を行うためのデバイスドライバがほとんど実装されていないので，ディスプレイドライバのような基本デバイスでさえも実機でテストを行うことは難しいでしょう．そのため，同時に配布開始になった「Virtual PC 2004」を用いてシミュレーションすることを勧めます．

筆者はDynabook G6/U22PDEW上(Pentium4 2.2GHz)にインストールし，簡単ながら動作検証を行えるようにしました．

● LonghornのVirtual PC 2004へのセットアップ

Virtual PC 2004へのインストールは，PC本体を1台用意するよりも簡単に行えます．また，今回のLonghorn評価版ではデバイスドライバが非常に少ないため，対応デバイスを探さなくても良いという利点もあります．また，MSDNの会員向けダウンロードサイトのコメントには，Virtual PC 2004用のVirtual Machine拡張プログラムがあると書かれているので，そちらのディスプレイデバイスを使用することで，インストール時に標準でVGA 16色(4ビット)に設定されるLonghornを，32ビット色などでテストすることが可能になります．

また，ディスプレイドライバが設定できない場合には，デバイスドライバの変更を手動で行い，Longhornに標準で付属するS3社のディスプレイデバイスドライバを強制的に指定することでも回避できるようです．

● Longhorn向け開発環境

Longhornでは，WinAPIそのものの扱いが変更され，Windowsカーネル自身が.NET IL(中間言語)化され，さらに多くのコンポーネントの分離統合が行われるようです．そのため，開発環境そのものも，現在のVisual Studio .NET 2003のマネージドクラスが中心になると思われます．

今回の開発者向け配布では，Longhorn SDKなどの開発キットが付属されていますが，コマンドラインベースのもので，GUI版はしばらく待つ必要があるようです．



Virtual PC 2004で起動したLonghorn

● Longhornのβ版配布開始までにできること

今回の配布では，Longhorn向けのアプリケーションを開発できるほどに環境が整っているわけではなく，また，β版の配布が開始されるまでに相当の手が加えられると考えられます．さらに，今回配布されているSDKは，おもにコマンドラインベースのものと，以前より提供されているILデバッガなどなので，β版でのテストに向けて一般的なアプリケーションを開発するのは難しいでしょう．

現在できることは，GUIを使用したクライアントアプリケーション，ミドルウェアなどで，マネージドクラスから使用できるコードに書き直すか，新たに設計を開始し，Visual Studio .NET 2003の.NET ILコードでのデバッグ，および開発をしておくことでしょう．

しかしながら，新機能のXAMLによるアプリケーション開発などのテストもできるようなので，以前のソフトウェアからの移行そのものを考えるよりは，現在は「新機能をどのように使いこなすか」といった点などに焦点を絞り，今後の開発に向けた基礎固めを行っておくべきかと考えています．

- 開発者向けLonghorn公開ページ
  http://msdn.microsoft.com/longhorn/
- Longhorn評価版セットアップガイド(英語)
  http://msdn.microsoft.com/longhorn/understanding/documentation/releasenotes/default.aspx

ひろはた・ゆきお OpenLab.

TEMPLATE PROGRAMMING

簡潔に，美しく，そして高品質に

C++テンプレートプログラミングの世界

特集

C++の能力を拡張するライブラリとしてテンプレートライブラリが広まりつつある．テンプレートライブラリは，たとえばプログラマが頻繁に使うデータ構造であるリスト構造を簡潔に記述でき，さらにそれを走査し，ソートするなどの機能をもつ．これによりプログラムを短くするだけでなく，実装時のバグを減らすという多大な効力が期待できる．

テンプレートライブラリの実装系としては，すでに多く使われているSTLにくわえ，最近ではBoostが注目を集めている．Boostはテンプレートライブラリとして優れており，ユーザー数も増加しつつあり，STLに次ぐ第2位の座を占めることも予想される．

そこで今回の特集では，これらC++で使えるテンプレートプログラミングについて解説する．また，テンプレート機能のないC言語でも，マクロやライブラリを用いることにより，テンプレートライブラリで提供されるコンテナ機能を実現できる．C言語プログラマにも本特集は見逃せない．



Chapter 1

C++プログラムを簡潔に，美しく，そして高品質に

テンプレートプログラミングの世界

宮坂 電人

C言語では，繰り返し記述を簡潔に行うためにマクロが使われてきた．しかしマクロは，ときに予期していない副作用が現れることがあり，その使用には十分な注意が必要とされる．これに対してC++言語では“テンプレート”の概念が導入され，型を意識しないプログラミングが可能なほか，コンパイル時に挙動を固定できるために，コードの実行効率が向上することも期待できる．

そこで本章では，今回の特集を概観すべく，テンプレートの概念を短いサンプルプログラムを交えながら解説する．（編集部）

テンプレートプログラミングとは文字どおり「テンプレートを使ったプログラミング」のことです．しかし，それだけでは従来のプログラミングとどう違うのか，わざわざ習得するメリットがあるのかどうか，はっきりしません．そもそもテンプレートを使うと，どのようなメリットがあるのか，あるいは問題点が解決されるのか，そこが気になるところでしょう．

そこで本章ではテンプレートの基本的な考え方や，テンプレートを使うことで従来のプログラムでは難しかった実装がいかに楽にできるかを説明していきましょう．また本特集で取り上げているSTLやBoostなどの背景としてかかわってくる，ジェネリックプログラミングの考えなども説明します．

〔リスト1〕似通った記述のあるコード(1)

```
static void Test()
{
    int a1,a2,a3;

    a1 = f();
    if(a1 < 0){
        a1 = -a1;
    }
    a2 = g();
    if(a2 < 0){
        a2 = -a2;
    }
    a3 = h();
    if(a3 < 0){
        a3 = -a3;
    }
    if(a1 < a2){
        a1 = a2;
    }
    if(a1 < a3){
        a1 = a3;
    }
    printf("max absolute value = %d¥n",a1);
}
```

同じ記述への対応

プログラミングをするとかならず「似通った記述」や「同じ記述」に出会うものです．たとえば，リスト1のようなC言語でなされたコーディングを見てみましょう．

見るからに要領が悪そうなコーディングがされていますが，すぐに気づくのは，

```
if(x < 0){
    x = -x;
}
```

という記述パターンと，

```
if(x < y){
    x = y;
}
```

という記述パターンが見受けられ，それが鼻につく点です．最初のパターンは「絶対値にする」ですし，次のは「二つのうち大きい値にする」であることがわかります．プログラミングの基本的な戦術として「似通ったパターンや同じパターンがあれば共通化する」があります．難しい言葉が好きな人なら「抽象化」とでも表現するところでしょうか[注1]．具体的には，

1) サブルーチンにする

2) マクロにする

あたりがC言語レベルでは常套（じょうとう）手段でしょう．

サブルーチンの導入

さきほどのプログラムで共通する記述パターンはリスト2のような二つの関数にできます．これらを利用して，さきほどの

注1：厳密にいうと「共通化」と「抽象化」は別物で，共通化は表面的な類似をまとめてしまう行為なのに対し，抽象化とは本質的なものをまとめてしまう行為を意味する．

〔リスト2〕共通部分を関数にする(1)

```
int abs_int(int x)
{
    if(x < 0){
        x = -x;
    }
    return x;
}
```

```
int max_int(int x,int y)
{
    if(x < y){
        x = y;
    }
    return x;
}
```

Test()を書き換えるとリスト3のようになります．最初の冗長な記述と比較すると，かなり行数が縮みました．もちろん，ここからさらに行数を縮めることも可能です．サブルーチンを導入することで見通しが良くなり，その結果，さらに見通しを良くするくふうを検討する余地ができました．

マクロの導入

しかし，こんな調子でなんでもかんでも共通部分をまとめられるわけではありません．その理由は"型"の存在です．たとえば，さきほどのTest()がリスト4のように記述されている場合はどうでしょうか．

最初のTest()と似通っていますが，int型で取り扱っていたものが，すべてdouble型に変更されています．ということは，せっかく作ったサブルーチンabs_intもmax_intもそのままでは使えません．型キャストをしても使えません．なぜなら，double型で表現している数値データはint型にキャストされる段階で小数点以下が切り捨てられるので使い物にならないからです．

となるとdouble型で使えるバージョンの関数を新たに作るハメになります．たとえば，リスト5のようになるでしょう．

しかしコードを見てわかるとおり，中身はabs_intとmax_intに似ている，というよりまったく同じ代物です．型が違うだけで，すでにある関数が流用できないという情けない状況です．

C言語ではこういうときにマクロを使うとよいのはご存知のとおりです．リスト6のようにすればいいわけです．こうしておくとabs_とmax_はint型だろうがdouble型だろうが流用できるはずです．

しかし注意しないといけないのは，マクロは単なる文字列の置き換えにすぎない点です．置き換えられた後のソースが期待しているものとかい離する可能性があります．たとえばリスト6のプログラムを縮めて，

```
a1 = abs_(f());
```

としたとします．もしもf()が2回以上呼び出したときに同じ値を返さない場合はどうなるでしょうか．この行はプリプロセッサを通過した後は，

```
a1 = (((f()) < 0) ? -(f()) : (f()))
```

となるため，期待した結果にはならないでしょう．

また，単なる置き換えであることを十分に意識し，演算子の優

〔リスト3〕共通部分の関数を利用する例

```
static void Test()
{
    int a1,a2,a3;

    a1 = f();
    a1 = abs_int(a1);
    a2 = g();
    a2 = abs_int(a2);
    a3 = h();
    a3 = abs_int(a3);
    a1 = max_int(a1,a2);
    a1 = max_int(a1,a3);
    printf("max absolute value = %d¥n",a1);
}
```

〔リスト4〕似通った記述のあるコード(2)

```
static void Test()
{
    double d1,d2,d3;

    d1 = df();
    if(d1 < 0){
        d1 = -d1;
    }
    d2 = dg();
    if(d2 < 0){
        d2 = -d2;
    }
    d3 = dh();
    if(d3 < 0){
        d3 = -d3;
    }
    if(d1 < d2){
        d1 = d2;
    }
    if(d1 < d3){
        d1 = d3;
    }
    printf("max absolute value = %lf¥n",d1);
}
```

〔リスト5〕共通部分を関数にする(2)

```
double abs_double(double x)
{
    if(x < 0){
        x = -x;
    }
    return x;
}

double max_double(double x,double y)
{
    if(x < y){
        x = y;
    }
    return x;
}
```

〔リスト6〕共通部分をマクロにする

```
#define abs_(X)  (((X) < 0) ? -(X) : (X))
#define max_(X,Y)  (((X) < (Y)) ? (Y) : (X))

static void Test()
{
    int a1,a2,a3;

    a1 = f();
    a1 = abs_(a1);
    a2 = g();
    a2 = abs_(a2);
    a3 = h();
    a3 = abs_(a3);
    a1 = max_(a1,a2);
    a1 = max_(a1,a3);
    printf("max absolute value = %d¥n",a1);
}
```

先順位にも注意する必要があります．マクロの記述で，やたらにカッコが多いのが目立つのはそのせいです．たとえば，abs_を，

```
#define abs_(X) (X < 0) ? -X : X
```

にしてもいいのではないかと思う人がいるかもしれません．一見問題がないように見えますが，もしも，

```
int a1 = 5;
int a2 = -10;
int a3 = abs_(a1+a2);
```

だったらどうでしょうか．この場合，最後の行は，

```
int a3 = (a1+a2 < 0) ? -a1+a2 : a1+a2;
```

と処理され，その結果，

```
int a3 = (5-10 < 0) ? -5-10 : 5-10;
```

となり，a3には期待している5ではなく，－15が入ります．

いうまでもなくabs_intやabs_doubleを使った場合は，このような不具合は生じません．

マクロは短い記述ですむのであれば便利ですが，長い記述は書くこと自体が苦痛ですし，後から検証やデバッグがやりにくいのも問題です．そのせいかどうかは知りませんが，C言語のキャリアが長い人でも，意外とマクロの使用頻度が少なかったりと，マクロが苦手な人は珍しくありません．

テンプレート

ここまで述べたC言語での二つの共通化テクニック，すなわち共通コードの関数化とマクロはいずれも便利な反面，限界を持っています．関数化では型をパラメータ化できないという限界がありますし，マクロは型の問題から逃れられても単なるソースの置き換えにすぎないという限界があります．

しかし，C++になると関数化とマクロの問題を解決するテンプレートというしくみが利用できます．テンプレートを一言で説明するなら，

● 型をパラメータ化できるしくみ

です．しかし後で説明しますが，型だけではなく，予想もしないものまでパラメータ化できるため，C言語はもちろんのこと，C++のエキスパートを自負する人にも想像がつかない意表をついた使い方ができます．C++を使っているのに，まるで別のプログラミング言語を使っているような錯覚を感じるほどです．

話を戻して，さきほどのabs_，max_をテンプレートで書き直したabsolute，maximumをリスト7に示します．

Tというシンボルはパラメータであり，ここがintであればint型用，doubleであればdouble型用になります．Test()はリスト8のように書き換えられます．

ここでテンプレートの基本的な書式を見てみましょう．テンプレートはその開始を意味する「template」で始まり，直後に「<」と「>」で囲んだ部分に必要なパラメータを記述します．パラメータが複数あるのなら「，」で区切ります．注意してほしいのは，パラメータを指定するときに「class」を前につけていますが，これは必ずしもパラメータがクラスであることを意味しません[注2]．記述上の約束でしかありません．事実，リスト8で示している例ではint型を指定していますが，いうまでもなくintはクラスではありません．

パラメータの記述の後にclassまたは関数が続きます．つまりテンプレートには，

● クラステンプレート
● 関数テンプレート

の2種類があることになります．

一方，テンプレートを利用する側は，テンプレートにパラメータを与えて実体化させる必要があります．これを「テンプレートのインスタンス生成[注3]」と称します．このときに，どの型で生成すべきかを指示するためにテンプレート名に続けて「<」と「>」で囲んだ部分に型を指定します．

なお，テンプレートに関する詳細な仕様については「プログラミング言語C++第3版」の第13章「テンプレート」も参照してください．

改良されたマクロ

テンプレートは「改良されたマクロ」という見方もできます．

〔リスト7〕共通部分をテンプレートにする

```
template <class T>
T absolute(T iD)
{
    return (iD < 0) ? -iD : iD;
}

template <class T>
T maximum(T i1,T i2)
{
    return (i1 < i2) ? i2 : i1;
}
```

〔リスト8〕テンプレートを利用する例

```
static void Test()
{
    int a1,a2,a3;

    a1 = f();
    a1 = absolute<int>(a1);
    a2 = g();
    a2 = absolute<int>(a2);
    a3 = h();
    a3 = absolute<int>(a3);
    a1 = maximum<int>(a1,a2);
    a1 = maximum<int>(a1,a3);
    std::cout << "max absolute value = " << a1 << std::endl;
}
```

注2：C/C++でよく見受ける一つのシンボルに複数の意味を持たせてしまう悪しき伝統(？)であろうか．最新の規格ではclassだけではなくtypenameも使えるので混乱するのが嫌な人はそちらも検討すべきと思われる．

注3：template instantiation.「プログラミング言語C++第3版」のC.13.7「インスタンス生成」を参照．

マクロの特徴である，
● 同じことの繰り返しをまとめられる
● パラメータを指定することでバリエーション(変型)を作りやすくなる
● 実行時ではなくコンパイル時に挙動を固定できるので，できあがったコードの実行効率が向上する
といったメリットが期待できます．その反面，
● どのようなコードに展開されるか予測がつかないことがある
● エラーメッセージが難解で，どこで記述を失敗したのか簡単にわからないことがある
● できあがったコードのサイズが膨れ上がる
というデメリットも出てくるので，かならずしも手放しで喜べない場合があります．

エラーメッセージが難解になるのは，エラーが発生するのがテンプレートを使用した箇所ではなく，コンパイラがコードを展開しようとした箇所，つまりテンプレートの実装部分，テンプレートの利用者が関知していない部分だからです[注4]．

できあがるコードのサイズが増えるのは指定したパラメータごとにコードが作られるせいで，これはテンプレート自体の特性なので，根絶することはおそらく不可能に近いでしょう．

さきほど，キャリアが長い人でも意外とマクロの使用頻度が少なかったり，マクロが苦手な人は珍しくないと述べたとおり，マクロやテンプレートのプログラミングはある意味，通常のプログラミングのセンスとは別のセンスを要求されます．下手をするとプログラミングではなくて，パズルあるいは手品をやっているような錯覚にとらわれることさえあります．

また絶対に勘違いしてほしくないのですが，C++の仕様だからといってテンプレートはオブジェクト指向だとは思わないことです．どちらかというと関数パラダイムに考えや感触が近い感じです．

ジェネリックプログラミング

C++のテンプレートを使ったプログラミングを「ジェネリックプログラミング(generic programming)」と称することがあります．C++の参考書を読むとよく出てくるのですが，ジェネリックプログラミングの明確な定義をあまり目にしません．genericを英和辞典で引くと「形容詞– 全般的な，包括的な」という漠然とした意味です．おおざっぱにいえば，
● 特定の型に依存しない汎用的な記述を目ざしたプログラミング
とでもいえばいいでしょうか．ちなみにC++の原作者であるBjarne Stroustrup氏のサイト[注5]にある用語集[注6]では，

> programming using templates to express algorithms and data structures parameterized by datatypes, operations,and polices.
> (訳)データタイプ，操作，ポリシ(注：policesはpolicyの複数形)がパラメータ化されたアルゴリズムやデータ構造を表現するためにテンプレートを使うプログラミング

と書かれています．つまりパラメータ化される対象として型だけではなく，他の要素も考慮されているわけです．このことを頭の中に置いておかないとジェネリックプログラミングがされているソースを読むときに混乱します．また自分でジェネリックプログラミングを目指したソースを書いたつもりでも，真の意味でジェネリックでないために，誰からも利用されないという悲しい結果になることもあります．

関数テンプレートの引き数の推定

ここからはテンプレートプログラミングやジェネリックプログラミングを実装するために利用するC++の仕様(ほとんどの入門記事では取り上げない範ちゅう)を紹介します．というのもテンプレートプログラミングを利用しているソースを読むと頭をかかえるような記述でついていけなくなる恐れがあるからです．

さて，さきほどのabsoluteを使って今度はdouble型に対応した記述をしてみましょう．単純に「absolute<int>」と書いたのを「absolute<double>」に書き換えてもいいのですが，リスト9のように型の指定を省略してもかまいません．というのも関数テンプレートの引き数が指定されていない場合，コンパイラが自動的に型を推定して当てはめるようになっているからです[注7]．

〔リスト9〕テンプレートの引き数の省略

```
static void Test()
{
    double d1,d2,d3;

    d1 = df();
    d1 = absolute(d1);
    d2 = dg();
    d2 = absolute(d2);
    d3 = dh();
    d3 = absolute(d3);
    d1 = maximum(d1,d2);
    d1 = maximum(d1,d3);
    std::cout << "max absolute value = " << d1 << std::endl;
}
```

注4：この点に関してはテンプレートの実装者自身も気づいていて，実装者側から対策を試みている場合もある．たとえばBoostではBoost Concept Check Libraryという試みがある．
注5：http://www.research.att.com/~bs/
注6：Bjarne Stroustrup's C++Glossary．http://www.research.att.com/~bs/glossary.html
注7：「プログラミング言語C++第3版」のC.13.4「関数テンプレートの引数の推定」を参照．

関数テンプレートの多重定義

コンパイラが自動的に推定するという仕様は恐ろしいことに通常関数にまで及んでいるため，関数テンプレートと通常関数を混ぜることも可能です[注8]．たとえば，すでにabsoluteという関数があったとします．これとテンプレート版のabsoluteが混ぜて使えるわけです（リスト10）．

ただし，筆者個人の意見ですが，多重定義は，あとからソースを検証する段階で目的の関数を取り違えてしまったり，バグの原因にもなりうるので濫用はつつしむべきかと思います．

〔リスト10〕テンプレートと通常関数の混用

```
int absolute(int iD)
{
    std::cout << "function version¥n";
    return (iD < 0) ? -iD : iD;
}

template <class T>
T absolute(T iD)
{
    std::cout << "template version¥n";
    return (iD < 0) ? -iD : iD;
}

static void Test()
{
    int a1,a2;
    double d1,d2;

    a1 = 123;
    d1 = -123.456;

    d2 = absolute(d1);      //template版が呼ばれる
    std::cout << "d2 = " << d2 << std::endl;
    a2 = absolute(a1);      //関数版が呼ばれる
    std::cout << "a2 = " << a2 << std::endl;
    a2 = absolute<int>(a1); //template版が呼ばれる
    std::cout << "a2 = " << a2 << std::endl;
}
```

〔リスト11〕複素数の絶対値を得る

```
#include <cmath>

class Complex {
    double mReal,mImage;
public:
    Complex(double iReal = 0,double iImage = 0){
        mReal = iReal;
        mImage = iImage;
    }
    double getReal() const {
        return mReal;
    }
    double getImage() const {
        return mReal;
    }
    double getAbs() const {
        return std::sqrt(mReal * mReal + mImage * mImage);
    }
};

Complex absolute(Complex iC)
{
    return Complex(iC.getAbs(),0);
}
```

ユーザー定義の特殊化

特定の引き数のときに実装を変えたいという状況はありがちです．absoluteの例でいえば複素数の絶対値を求める関数は「マイナスの数値なら再びマイナスにする」という求め方では不都合です．仮に複素数を扱うComplexクラスがあったとしましょう．この場合は，リスト11のようにComplex型のabsoluteを提供するところです．

しかし，提供された側がうっかりテンプレート版のabsoluteを利用したらどうなるでしょうか．この場合，エラーになったり，あるいは予想しない結果が起きてしまうでしょう．これを防止するため「ユーザー定義の特殊化」[注9]を使います．たとえば，リスト12のように記述します．ユーザー定義の特殊化は「template<>」という記述に続けて，定義したいコーディングを続けます．

ユーザー定義の特殊化はエラー回避やバグ対策に限ったものではなく，特定のパラメータでの例外的な記述，実行効率の向上，コードサイズの肥大化対策[注10]によく使われます．

型以外のテンプレートの引き数

通常，テンプレートの引き数としてclass（あるいはtypename）によって型を指定させますが，intなどの通常の型の引き数や，テンプレート引き数も指定できます[注11]．

また頻繁に使いそうな引き数をデフォルトとして，あらかじ

〔リスト12〕ユーザー定義の特殊化の例

```
template<> Complex absolute<Complex>(Complex iC)
{
     //Complex absolute(Complex iC)を呼び出す
    return absolute(iC);
}

static void Test()
{
    Complex c1(1,1);
    Complex c2;

    c2 = absolute(c1);          //OK
    std::cout << "c2#real = " << c2.getReal() << std::endl;
    c2 = absolute<Complex>(c1); //こちらもOK
    std::cout << "c2#real = " << c2.getReal() << std::endl;
}
```

注8：「プログラミング言語C++第3版」の13.3.2「関数テンプレートの多重定義」を参照．
注9：user-defined specialization.「プログラミング言語C++第3版」の13.5「特別バージョン」を参照．訳書では「ユーザー定義の特別バージョン」と翻訳されている．
注10：「プログラミング言語C++第3版」の13.5「特別バージョン」にvoidポインタバージョンを利用することでコードの肥大化を抑えるくふうを紹介している．
注11：「プログラミング言語C++第3版」の13.2.3「テンプレート引数」を参照．

〔リスト 13〕型以外のテンプレートの引き数

```
void NoneCheck(int);

typedef void (*IndexChecker)(int);

template <class TYPE,int MIN,int MAX,IndexChecker IC = NoneCheck>
class FixedArray
{
    TYPE mArray[MAX - MIN + 1];
public:
    TYPE& operator[](int index){
        IC(index);
        return mArray[index - MIN];
    }
};

void NoneCheck(int)
{
}
```

め指定することもできます[注12]．リスト 13に示すのは，その実例です．

ここで示している FixedArray は固定した配列を実現するものですが，通常の配列とは違い，0以外からインデックスを開始できるようにし，さらにインデックス値をチェックできるようなしくみを埋め込んでいます．MIN はインデックスの開始値，MAX は終了値で，それぞれ int で指定できます．IndexChecker はインデックス値チェック関数を指定する関数ポインタの型です．「IndexChecker IC=NoneCheck」となっているのは，もしも3番目の引き数に何も指定しないなら NoneCheck という関数(まったくチェックをしない関数)がデフォルトで指定されることを意味します．これを使う例はリスト 14のとおりです．

CheckHarder はテンプレート関数で，ここでも int で範囲を指定しています．こんなことをせずに，

```
void CheckHarder(int index,int MIN,int MAX)
```

としてもかまわないのですが，変数ではなく定数にすることで実行効率が上がる場合があります．

つまり，テンプレートのメリットとしてコンパイル時に定数を埋め込むことができ，その結果，実行効率の向上が期待できるということです[注13]．

ポリシの指定

テンプレートプログラミングあるいはジェネリックプログラミングで「ポリシ」という言葉を使う場合があります．どういう意味かを明確に説明している例が少ないのですが，しいていうなら，

● 挙動の指定を(実行時にではなく)コンパイル時にするもの

でしょう．デザインパターンで Strategy パターンというのがありますが，これはアルゴリズムの指定を動的に変更できるパターンです．

〔リスト 14〕FixedArray の使用例

```
template <int MIN,int MAX>
void CheckHarder(int index)
{   //index が MIN から MAX の範囲になければ
    //エラーメッセージを出して処理を中断する
    if(index < MIN){
        std::cerr << "index:" << index
                  << " is less than MIN:" << MIN << std::endl;
        std::abort();
    }
    if(index > MAX){
        std::cerr << "index:" << index
              << " is bigger than MAX:" << MAX << std::endl;
        std::abort();
    }
}

int main()
{
    int aIndex;
    std::cout << "* start *¥n";
    //-10 から 10 までのインデックスを取る int 型固定配列
    FixedArray<int,-10,10> aFA;
    for(aIndex = -10; aIndex <= 10; aIndex++){
        aFA[aIndex] = aIndex;
    }
    for(aIndex = -10; aIndex <= 10; aIndex++){
        std::cout << "aFA[" << aIndex << "]=" << aFA[aIndex]
                  << std::endl;
    }
    //-9 から 9 までのインデックスを取る int 型固定配列
    FixedArray<int,-9,9,CheckHarder<-9,9> > aFB;
    for(aIndex = -9; aIndex <= 10; aIndex++){
        aFB[aIndex] = aIndex;
    }

    std::cout << "* end *¥n";
}
```

たとえば，ある作業をするクラスがあったとします．そのクラス内でエラーがあったときに，例外を出すべきか，何もなかったと無視すべきか，何らかのハンドラを呼ぶのかをあらかじめ固定できない場合があります．

それとは別の要求で，マルチスレッド内で使うので複数スレッドが通過できないようにしたい場合，OSによってスレッドをせきとめるしくみが変わってしまうので，あらかじめコードを埋め込めない場合があります．あるいはシングルスレッドで使うつもりだから，最初からそんな余計なものを埋め込んでほしくないという要求もあります．

この場合，二つの Strategy があります．つまり，

● エラー時の挙動
● スレッド対策の挙動

です．それぞれの挙動の切り替えをプログラム内で律儀に記述するとたいへんなことになります．もしもエラー時の挙動が A 通り，スレッド対策の挙動が B 通りあると，単純計算で $A \times B$ 通りの内部分岐が出てきます．この調子で切り替えるべき挙動が増えていくと，組み合わせの爆発現象で頭をかかえるハメに

注 12：「プログラミング言語 C++ 第3版」の 13.4.1「デフォルトテンプレート引数」を参照．
注 13：ただし，本当に向上するかは実際にテストしないと，どうともいえない場合もある．

〔リスト 15〕ポリシを埋める前の実装

```
//TYPEはリングバッファに記録するデータの型,
//SIZEはリングバッファのサイズ
template <class TYPE,int SIZE>
class RingBuffer
{
    TYPE mArray[SIZE];
    size_t mCount; //リングバッファに格納している個数
    //リングバッファへの記録インデックス
    int mPushIndex,mPopIndex;
public:
    RingBuffer(){
        mCount = 0;
        mPushIndex = mPopIndex = 0;
    }

    void push(TYPE iData){ //リングバッファに登録する
        //リングバッファが満杯でないなら登録する
        if(mCount < SIZE){
            mArray[mPushIndex++] = iData;
            if(mPushIndex >= SIZE){
                mPushIndex = 0;
            }
            ++mCount;
        }
    }

    TYPE pop(){ //リングバッファから取り出す
        TYPE aAns;
        //リングバッファが空っぽでないなら取り出せる
        if(mCount > 0){
            aAns = mArray[mPopIndex++];
            if(mPopIndex >= SIZE){
                mPopIndex = 0;
            }
            --mCount;
        }else{ //空っぽなら0を返す
            aAns = 0;
        }
        return aAns;
    }
    //リングバッファに登録している個数をえる
    size_t count() const {
        return mCount;
    }
};

int main()
{
    std::cout << "* start *¥n";
    RingBuffer<int,5> aRB;
    int aIndex;
    for(aIndex = 0; aIndex < 7; aIndex++){
        aRB.push(aIndex + 1);
        std::cout << "count = " << aRB.count() << " ";
    }
    std::cout << std::endl;
    for(aIndex = 0; aIndex < 10; aIndex++){
        std::cout << "count = " << aRB.count() << " ";
        std::cout << "aIndex:" << aIndex << " = " <<
        aRB.pop() << std::endl;
    }
    std::cout << "* end *¥n";
}
```

なります．また，現時点では未知だが将来的に挙動が追加される場合もあります．現時点で出荷されていないOSでどう処理すべきかなど事前に決められませんし，もちろんプログラムに埋め込むことなど不可能です．

継承や多重継承を使ったアプローチも考えられます．ただし複数の挙動を実装するとなると，デザインパターンでいうところのDecoratorパターンとして解釈しないとまずい状況もありえます．この場合，C++では継承や多重継承を使うと，かえってややこしくなったり言語仕様的に実装できないこともあります．

ところがテンプレートプログラミングでは変化するであろう挙動をパラメータ化することで，今まで述べたような悩みが解決します．変化するであろう挙動(ポリシ)を未知のままにしておき，それをプログラムに埋めてしまうのです．

また，ポリシをパラメータ化することでコードがスリムになったり実行効率の向上が期待できます．というのも，あらゆる挙動をプログラムに盛り込む必要がないのでコード量はその分減ります．$A \times B$ 通りを埋め込んだとしても使うパターンが少ない場合もあります．その場合，実際に使うパターン分だけテンプレートのインスタンス生成をすればいいので，かえってテンプレートのほうがコード量が少なくてすみます．余計な条件分岐処理がなくなるため実行効率が向上することも期待できます．

ここでポリシを指定させることでプログラミングがどう変化するかを実際のコードで見てみましょう．リスト 15に示すのはリングバッファをテンプレートで実現したものです．見てわかるとおり，基本的にはpushでリングバッファに登録し，popで登録したものを取り出すという単純なものです．しかし，ここで注文がつけられました．

- エラー時の処理が甘い：現状ではリングバッファが満杯のときにpushを行っても無視，空っぽのときにpopを行うと0を返すだけだが，それぞれの状況で例外を発生させるようにしてほしい．
- マルチスレッド対策がない：現状ではリングバッファを読み書きしている途中で複数スレッドの通過を許しているので，このままだとマルチスレッド環境では使いものにならない．

注文を検討してみると，いずれも問題点があります．例外を発生するにしても，どのような種類の例外を発生させればいいかは状況によって変わってくるでしょうし[注14]，例外を拒絶する人もいます．マルチスレッド対策するにしてもOSによって呼び出す処理が変わりますし，シングルスレッドでしか使わない人からは余計な処理を埋めるなと苦情がくるでしょう．

いずれにせよ，あらかじめ注文を具体的なコードとして埋め込めません．そこでそれぞれの注文をポリシとして外部から指定できるようにします．利用する側は自分たちに都合のいいポリシを指定して使えばいいわけです．

まずエラー処理のポリシはリスト 16のようにします．まったくエラー処理をせず素通りさせるつもりなら，リスト 17のような実装になります．もしも例外を発生させるつもりならリスト 18の実装になります．

一方，マルチスレッド対策のポリシはリスト 19のようにし

注 14：特定のクラスライブラリを併用していて，例外はそのクラスライブラリが定義しているものを利用したい場合など．

〔リスト 16〕エラー処理のポリシ

```
template <class TYPE>
class エラー処理ポリシ {
public:
    static void full(){
        ...(満杯時のエラー処理を行う)...
    }
    static TYPE empty(){
        ...(空っぽな時のエラー処理を行う)...
    }
};
```

〔リスト 17〕エラーを無視するポリシ

```
template <class TYPE>
class RBNoCheck {
public:
    static void full(){
    }
    static TYPE empty(){
        return 0;
    }
};
```

〔リスト 18〕例外を放出するポリシ

```
template <class TYPE>
class RBThrowRTErr {
public:
    static void full(){
        throw std::runtime_error("ring buffer is full");
    }
    static TYPE empty(){
        throw std::runtime_error("ring buffer is empty");
    }
};
```

〔リスト 19〕マルチスレッド対策のポリシ

```
class マルチスレッド対策ポリシ {
public:
    void Lock(){
        ...(ロックする，ここから通過できるのは
            一つのスレッドのみとする)...
    }
    void Unlock(){
        ...(ロックを無効化する)...
    }
};
```

ます．シングルスレッドだけでかまわないならリスト 20の実装で十分でしょう．

一方，オリジナルOSで以下のようなAPIがあった場合注15は，

● LockHandle NewLockHandle()：ロックオブジェクトを新規に確保する
● void DisposeLockHandle(LockHandle handle)：確保したロックオブジェクトを解放する
● void LockObject(LockHandle handle)：ロックをかけて単一スレッドのみを通過させる
● void UnlockObject(LockHandle handle)：ロックオブジェクトを無効化する

となり，具体的にはリスト 21のような実装になります．

このようにしてエラー処理ポリシとスレッド対策ポリシをパラメータ化したもので，さきほどのリングバッファクラスを実装し直すとリスト 22のようになります．

```
RingBuffer<int,5,RBThrowRTErr<int>,
                              OrigOSTheadLock>
```

という記述が長くて，いちいち記述するのがめんどうな場合は typedef を利用するとよいでしょう．テンプレートを利用するコードではわりあいによく見かける記述テクニックです．

ポリシを指定する方法はよく考えると Template Method パターンのように注文のベースとなる抽象クラスを決めておき，具体的な注文を具象クラスで実装してもいいように思います．その場合，仮想関数の呼び出し分のコストがかかりますし，テンプレートよりも実装に若干手間どります．テンプレートを使った場合はコンパイル時に実行コードが固定されるので余計なオーバヘッド（つまり判断分岐や仮想関数の呼び出しなどの）はなくなります．

〔リスト 20〕シングルスレッドで利用するポリシ

```
class RBNoThread {
public:
    void Lock(){}
    void Unlock(){}
};
```

〔リスト 21〕オリジナルOSで利用するポリシ

```
class OrigOSTheadLock {
    LockHandle mLockHandle;
public:
    OrigOSTheadLock(){
        mLockHandle = NewLockHandle();
    }
    ~OrigOSTheadLock(){
        DisposeLockHandle(mLockHandle);
    }
    void Lock(){
        LockObject(mLockHandle);
    }
    void Unlock(){
        UnlockObject(mLockHandle);
    }
};
```

アダプタとしてのテンプレート

すでに存在する実装をベースにして，そこに少しの追加実装を加えるだけで，まったく一から作らなくてすむ手法をアダプタあるいはラッパと呼ぶことがあります．テンプレートを使ったプログラムはアダプタやラッパ的な実装にも応用できます．

たとえば，リスト 23のようなリングバッファのクラスが用意されていたとします．このクラスは汎用性を考えたため void ポインタを利用していますが，実際のプログラムでは型キャストによって適合させねばなりません．しかし，型キャストはコンパイラに対する「自己申告」なので，間違った申告がなされて

注15：ここで書いているオリジナルOSは架空のもので実在しない．あくまで例として書いていることに注意．

〔リスト 22〕ポリシを埋めた後の実装

```
template <class LOCKER>
//RAIIイディオムを利用して関数の先頭でロックをかけ,
//関数から抜けると自動的にロックをはずすクラス
class LockRAII
{
    LOCKER& mLocker;
public:
     LockRAII(LOCKER& iLocker) : mLocker(iLocker) {
        mLocker.Lock();
    }

    ~LockRAII() {
        mLocker.Unlock();
    }
};

template <class TYPE,int SIZE,class ERROR = RBNoCheck<TYPE>,
                                      class LOCKER = RBNoThread>
class RingBuffer
{
    TYPE mArray[SIZE];
    size_t mCount;
    int mPushIndex,mPopIndex;

    LOCKER mLocker;
public:
    RingBuffer(){
        mCount = 0;
        mPushIndex = mPopIndex = 0;
    }

    void push(TYPE iData){
        LockRAII<LOCKER> aLock(mLocker);

        if(mCount < SIZE){
            mArray[mPushIndex++] = iData;
            if(mPushIndex >= SIZE){
                mPushIndex = 0;
            }
            ++mCount;
        }else{
            ERROR::full();
        }
    }
    TYPE pop(){
        LockRAII<LOCKER> aLock(mLocker);

        TYPE aAns;
        if(mCount > 0){
            aAns = mArray[mPopIndex++];
            if(mPopIndex >= SIZE){
                mPopIndex = 0;
            }
            --mCount;
        }else{
            aAns = ERROR::empty();
        }
        return aAns;
    }

    size_t count() const {
        return mCount;
    }
};

int main()
{
    std::cout << "* start *¥n";
    RingBuffer<int,5,RBThrowRTErr<int>,OrigOSTheadLock> aRB;
    int aIndex;
    try{
        for(aIndex = 0; aIndex < 7; aIndex++){
            aRB.push(aIndex + 1);
            std::cout << "count = " << aRB.count() << " ";
        }
        std::cout << std::endl;
        for(aIndex = 0; aIndex < 10; aIndex++){
            std::cout << "count = " << aRB.count() << " ";
            std::cout << "aIndex:" << aIndex << " = "
                      << aRB.pop() << std::endl;
        }
    }
    catch(const std::runtime_error& iErr){
        std::cerr << "runtime_error:" << iErr.what()
                  << std::endl;
    }
    std::cout << "* end *¥n";
}
```

〔リスト 23〕汎用性を考えたリングバッファの例

```
class RingBuffer
{
    typedef void* UnivPtr;
    UnivPtr* mArrayPtr;
    size_t mCount,mMaxCount;
    int mPushIndex,mPopIndex;

    RingBuffer(); //(サイズ指定を必ずさせるための工夫)
public:
    RingBuffer(size_t iMaxCount) {
        mArrayPtr = new UnivPtr[iMaxCount];
        mMaxCount = iMaxCount;
        mCount = 0;
        mPushIndex = mPopIndex = 0;
    }

    ~RingBuffer() {
        delete[] mArrayPtr;
    }

    bool isEmpty() const {
        return (mCount == 0);
    }

    bool isFull() const {
        return (mCount >= mMaxCount);
    }
    //登録できたらtrueを返す,できないならfalseを返す
    bool push(void* iPtr) {
        if(isFull()){
            return false;
        }else{
            mArrayPtr[mPushIndex++] = iPtr;
            if(mPushIndex >= mMaxCount){
                mPushIndex = 0;
            }
            ++mCount;
            return true;
        }
    }

    void* pop() {
        void* aAns;
        if(isEmpty()){
            aAns = NULL;
        }else{
            aAns = mArrayPtr[mPopIndex++];
            if(mPopIndex >= mMaxCount){
                mPopIndex = 0;
            }
            --mCount;
        }
        return aAns;
    }

    size_t count() const {
        return mCount;
    }
};
```

〔リスト24〕RingBufferを使った例

```
int main()
{
    std::cout << "* start *¥n";

    RingBuffer aRB(5);
    int aIndex;
    for(aIndex = 0; aIndex < 7; aIndex++){
        if(!aRB.isFull()){
            aRB.push(new int(aIndex + 1));
            std::cout << "count = " << aRB.count() << " ";
        }
    }
    std::cout << std::endl;
    for(aIndex = 0; aIndex < 10; aIndex++){
        std::cout << "count = " << aRB.count() << " ";
        int* aDP = static_cast<int*>(aRB.pop());
        int aD;
        if(aDP != NULL){
            aD = *aDP;
            delete aDP;
        }else{
            aD = 0;
        }
        std::cout << "aIndex:" << aIndex << " = " << aD
                  << std::endl;
    }
    while(!aRB.isEmpty()){
        int* aDP = static_cast<int*>(aRB.pop());
        delete aDP;
    }

    std::cout << "* end *¥n";
}
```

もコンパイラはそのまま通過させます．またvoidポインタを利用するということは，登録や取り出しはすべてポインタベースとなって使い勝手が良くありません．たとえばintを記録して取り出す例はリスト24のようになりますが，見てわかるとおり，何やら苦しい雰囲気です．というのもintデータを動的に確保してから記録，取り出したデータの解放，リングバッファ内にたまっているデータの解放処理（解放しないとメモリリークになるので）が付随するからです．

RingBufferをテンプレートベースに改造するのがベストですが，アダプタによる追加で解決してみましょう．というのもすでに存在する実装をまるまる書き換えていられない場合もありがちなので，その対策例を示したいからです．

まず型キャストの弊害を減らすためにvoidポインタのアクセスから任意のポインタによるアクセスができるようにしましょう．リスト25のようになります．これによってpopに型キャストをつけなくてすみますし，pushもきちんと型を認識するので，間違った型を渡すとエラーを報告するようになります．また見てわかるとおり，アダプタによるアプローチは，記述するコード量がかなり少なくなるというメリットがあります．少ない手間で実装できますしテストやデバッグの手間も減ります．

これでも使えなくはないですが，動的な確保や解放，メモリリークを予防するための解放処理がまだ必要です．リスト26のようにするとそれらが不要になります．

pushやpopにはポインタではなく実体（あるいはリファレンス）でアクセスできるようになります．またメモリリーク防止

〔リスト25〕アダプタによる追加の例（1）

```
template <class TYPE,int MAXSIZE>
class RingBuffer_Ptr : public RingBuffer
{
public:
    RingBuffer_Ptr() : RingBuffer(MAXSIZE){}

    bool push(TYPE* iPtr){
        return RingBuffer::push(iPtr);
    }

    TYPE* pop(){
        return static_cast<TYPE*>(RingBuffer::pop());
    }
};
```

〔リスト26〕アダプタによる追加の例（2）

```
template <class TYPE,int MAXSIZE>
class RingBuffer_NotPtr : public RingBuffer
{
public:
    RingBuffer_NotPtr() : RingBuffer(MAXSIZE){}

    ~RingBuffer_NotPtr(){
        while(!isEmpty()){
            delete pop_ptr();
        }
    }

    bool push(const TYPE& iRef){
        bool aResult = !isFull();
        if(aResult){
            TYPE* aData = new TYPE(iRef);
            RingBuffer::push(aData);
        }
        return aResult;
    }

    TYPE pop(){
        TYPE aResult;
        if(!isEmpty()){
            TYPE* aData = pop_ptr();
            aResult = *aData;
            delete aData;
        }
        return aResult;
    }

    TYPE* pop_ptr(){
        return static_cast<TYPE*>(RingBuffer::pop());
    }
};
```

のための解放処理はデストラクタで自動的になされるので利用者側では不要になります．これを使った例はリスト27のようになります．さきほどのRingBufferを使った例と比較すると行数が減り，わかりやすくなっています．

テンプレートプログラミングのメリット/デメリット

ここでテンプレートプログラミングのメリットを簡単にまとめてみましょう．

1) マクロと違いソースの単純置き換えによる予期しない弊害を予防しやすくなる
2) 型キャストの弊害を予防しやすくなる
3) 最終的に実行させたい条件を変数ではなく定数に決定でき

〔リスト27〕RingBuffer_NotPtrを使った例

```
static void test2()
{
    RingBuffer_NotPtr<int,5> aRB;
    int aIndex;
    for(aIndex = 0; aIndex < 7; aIndex++){
        if(!aRB.isFull()){
            aRB.push(aIndex + 1);
            std::cout << "count = " << aRB.count() << " ";
        }
    }
    std::cout << std::endl;
    for(aIndex = 0; aIndex < 10; aIndex++){
        std::cout << "count = " << aRB.count() << " ";
        int aD = aRB.pop();
        std::cout << "aIndex:" << aIndex << " = " << aD
                  << std::endl;
    }
}
```

ることで，実行効率の向上が期待できる

4) 最終的に実行させたいコードをコンパイル時に静的に決定できることで，実行時の分岐処理や継承，仮想関数によるオーバヘッドを予防しやすくなる

5) 現時点では記述できない実装を後から付け加える方策をとりやすくする

同じくデメリットも簡単にまとめてみましょう．

1) 利用も実装も従来のプログラミングと比較して難解になる場合がある

2) どのようなコードに展開されるか予測がつかないことがある

3) エラーメッセージが難解になる場合がある

4) パラメータの指定ごとにコードができてしまうためコードの肥大化をまねきやすい

おわりに

テンプレートプログラミングの導入編を紹介しましたが，いかがだったでしょうか．ベテランのC++プログラマでも案外テンプレートを知らなかったり，敬遠していることがありがちです．それはテンプレートを従来のプログラミング実装の延長線上でとらえようとして失敗していることも一因だと思います．途中でも述べましたがテンプレートを使ったプログラムは下手をするとパズルや手品に近いような感覚におそわれることがあります．また，いわゆるオブジェクト指向的な雰囲気とは微妙にくい違っています．

しかし，繰り返しを抑えたり検討事項を後回しにして上手に抽象化をはかるツールとしてテンプレートは重要ですし，強力な支援となってくれるでしょう．どうか毛嫌いせずにテンプレートを検討してもらいたいと思います．

本章を書くにあたりC++コンパイラは筆者が最近よく使うMac OS X 10.3上のGCC 3.3を利用しました．ほかのコンパイラを利用した場合，本章で示したとおりにならないケースがありうることにご注意ください．

みやさか・でんと　miyadent@anet.ne.jp

New Products ── Linear Technology，最大充電レートを保てるリチウムイオン電池充電用LSI「LTC4058」を発売
米Linear Technology社は，過熱を防ぎ常時最大充電レートを保てるリチウムイオン電池充電用LSI「LTC4058」を発売した．1セルが4.2Vのリチウムイオン電池を充電できる．実装面積は9mm²，充電電流は最大で800mA.

Chapter 2

コンテナから正規表現，関数オブジェクト，数学ライブラリまで

新世代テンプレートライブラリ Boostの全貌

矢野 越夫

C++でのテンプレートといえばSTL(Standard Template Library)が有名で，すでに幅広く使われている．しかしここに来て，STLで問題となっていた点を改良し，さらに機能を豊富にした新しいテンプレートライブラリBoostが注目を集めている．
Boostは型を保証した文字出力format()，正規表現regex_search()，STLでも使われるコンテナとイテレータ，各種データ構造やメモリ操作機能など，プログラミングにおいて頻繁に使われる機能が満載されている．それだけでなく，Qtに実装されている"シグナル&スロット"機構を提供するなど，先進的な機能も備えている．
また，当初からマルチプラットホーム展開が意識されており，「退行試験」と呼ばれる互換性チェック機能により，安心してライブラリを使えるようになっていることも目新しい．
そこで本章では，この新しいテンプレートライブラリBoostについて，その概要と使い方をサンプルプログラムを交えて解説する．（編集部）

Boostは C++の標準ライブラリを補強するため作られたライブラリ群である．多くの技術者により開発され，http://www.boost.org/でフリーで公開されている．そして，いくつかのライブラリは，やがてC++の標準となるべく提案されている．

ほとんどの機能がC++のテンプレートとして提供されているので，Boostはヘッダをインクルードするだけで気軽に使うことができる．ただし，正規表現ライブラリなどの大きな機能は別途ライブラリをリンクする必要がある．

Boostの機能はあまりにも多すぎて，ここですべてを紹介することはできない．そこで，本章ではSTL(Standard Template Library)を補強するという意味で，STLから大きく改良されている機能を中心に紹介することにする．

Boostのインストール

● Boost本体

Boostのインストールは比較的簡単である．ここではWindows XP上の開発環境，Microsoft Visual C++6.0(MSVC6)にインストールする方法を述べる．その他の環境もBoostのドキュメントに詳述されている．最新版であるVisual C++ .NET 2003(MSVC7.1)でも，ほとんど同様の手順でインストールできる．

▶ダウンロード

まず，http://sourceforge.net/project/showfiles.php?group_id=7586よりBoostのソースコードとドキュメントをダウンロードする．ただし，ディレクトリは変更されているかもしれない．http://www.boost.org/から最新のソースコードをたどったほうがよいだろう．

▶Boost本体の展開

本章ではBoost Version 1.30.2をダウンロードして動作を確認した．適当なディレクトリでZIPファイルを解凍すると，boost-1.30.2というディレクトリが生成される．以下，本章はD:¥boost-1.30.2¥にBoostが展開されているものとする．

▶ライブラリの内訳

Boostにはヘッダだけで動作するテンプレート機能と，別途リンクライブラリ[注1]をリンクする必要がある関数とが混在している．リンクライブラリが必要な機能を表1に示す．

テンプレート定義しか使わないのであれば，ライブラリをコンパイルする必要はない．単にヘッダをインクルードするだけで動作する．

● Boostライブラリのコンパイル

Boostは Bjam(Boost jam)と呼ばれる生成ツールで全体をコンパイルできる．

▶Bjam.exe

Bjam.exeは，http://prdownloads.sourceforge.net/boost/bjam-3.1.4-2-ntx86.zip?downloadにある．筆者のWindows XP環境ではそのまま問題なく動作した．

Bjam.exeが動作しないときには自分でコンパイルする必要がある．環境変数INCLUDEとLIBを検査して，何も設定されていないことを確認する．もし，MSVC7.1も同時にインストールしてる環境ならば，すでに環境変数が設定されているため，

〔表1〕リンクライブラリが必要な機能

date_time
filesystem
graph
python
regex
signals
test
thread

注1：Boostでは#inculdeするヘッダファイル(*.h)も，リンクを行うオブジェクトファイル(*.o，*.so)もBoostライブラリと呼んでいる．ここでは後者のことをリンクライブラリと呼び，区別している．

vcvars32.bat を実行しても MSVC6ではコンパイルできない．この場合は，MSVC7.1関連の環境変数を消去するか，MSVC7.1を使う．

次に Bjam のコンパイル手順を示す．

1) コマンドプロンプトを開く
2) MSVC6の vc98¥bin¥vcvars32.bat を実行
3) D:¥boost-1.30.2¥tools¥build¥jam_src¥にカレントディレクトリを移動する
4) build.bat を実行

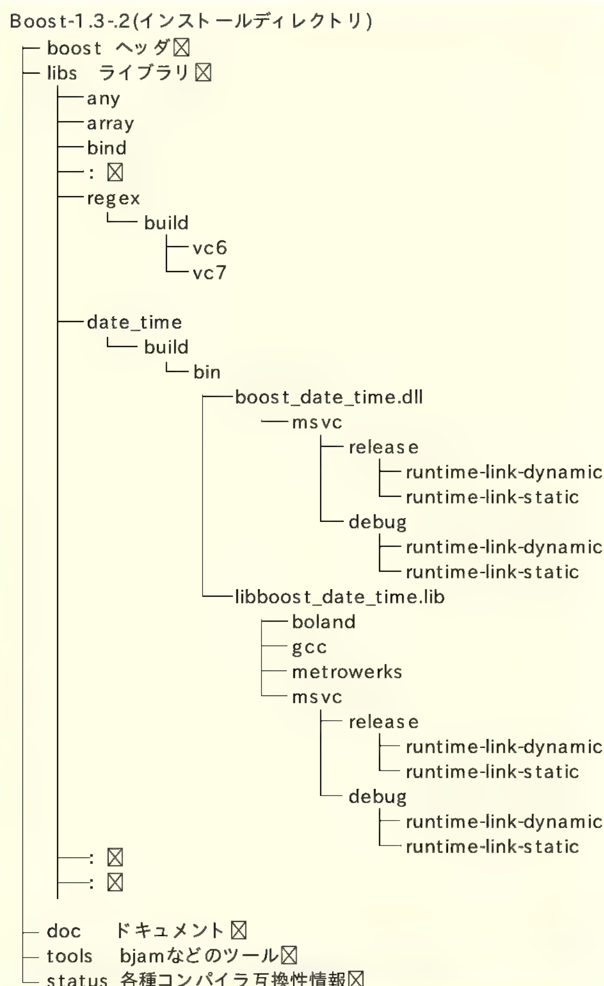
なお，MSVC7.1ではそのままコンパイルできなかった．build.bat と build.jam の中の cl のオプション /LIBPATH が間違っているので，削除することによりコンパイルできる．

▶Boostライブラリのビルド

bjam.exe を D:¥boost-1.30.2¥にコピーする．カレントディレクトリを D:¥boost-1.30.2¥に移動して，次のようにして Boostライブラリをビルドする

```
bjam "-sTOOLS=msvc"
        "-sBUILD=release<runtime-link>static"
```

〔図1〕Boostのディレクトリ構造



この例では release 版の静的ライブラリを作る．デバッグ版を作るなら release の代わりに debug とし，動的ライブラリを作るなら static のところを dynamic とする．

だいたいは Bjam コマンド一発でコンパイルしてくれるのだが，コンパイルされないモジュールもある．

regex は Bjam で途中までコンパイルしているようだが，どうも完全には動作しない．そこで，次に MSVC6における regex の単独コンパイルとインストールを示す．

1) コマンドプロンプトを開く
2) MSVC6の vc98¥bin¥vcvars32.bat を実行
3) D:¥boost-1.30.2¥libs¥regex¥build¥にカレントディレクトリを移動
4) nmake-fvc6.mak
5) nmake-fvc6.mak install

これで regex が MSVC6から利用可能になる．5)の install でライブラリをすべて MSVC6の VC98¥lib¥の下にコピーする．MSVC7.1環境のときは，vc6.mak の代わりに vc7.mak を使う．VC7の vcvars32.bat は Common7¥Tools の下にある．

5)の nmake install を実行するとコンパイラの lib の下に各ライブラリすべてをコピーする．コード中にライブラリを指定するなら実行する必要はない．

▶ディレクトリ構造

Bjam でビルドを終えたときのディレクトリ構成を図1に示す．

まだ整理されていないようで，バラバラの感が強い．bin-stage の下にまとめられているのもあるが，date_time のように奥底に作られるライブラリもある．

● MSVC の設定

▶インクルードパス

Boost のサンプルは #include <boost/xxxx> と記述されているので，MSVC6の“ツール”メニューの“オプション”ダイアログを開き“ディレクトリ”タブの“インクルードファイル”に，Boost インストールパスを追加する．プロジェクトファイルを共有化するなら，ここに設定せずに，コンパイルオプションの /I を使ったほうがよい．

▶ライブラリパス

ライブラリの存在場所が一定してないので，機能単位にフルパスを書くしかない．ビルド環境を共有するためには，プロジェクト設定でリンクライブラリに設定するのがよい．もしくは必要なライブラリを VC98¥lib¥にコピーするとめんどうがなくなる．それとも共有サーバに Boost ライブラリ全体を入れておいて，#pragma comment にてソースコード中に書くのも悪くない．

ライブラリはどこか1か所にまとめたほうが使いやすいと思うが，現在の Boost ビルド環境ではサポートされていない．将来的にディレクトリ構造が変更されるかもしれないので，その都度コピーするしかないだろう．

▶ プロジェクト設定

リンクライブラリが必要なプロジェクトは，MSVC6の“プロジェクト設定”ダイアログを開き，“リンク”タブの“インプット”カテゴリの中で，“追加ライブラリのパス”に Boost のパスを加える．本章ではわかりやすくするためコード中に `#pragma comment` で指定している．

● 退行試験

Boost ライブラリには，すべての機能がプラットホームに合致するかどうか試験するスクリプトが含まれている．これも Bjam を使って実施することができる．これを退行試験と呼んでいる．退行試験は，コンパイルエラーが出たとき，記述ミスなのか，もともとコンパイルできないテンプレートなのかの区別に役立つ．

▶ 退行試験の実施

実際にはアーカイブ配布時にすでに配布者によって結果が用意されているので，実施する必要はない．完全な退行試験結果テーブルを得るには GCC や Boland C++ などのコンパイラも利用できる環境でなければならない．さらに，ログの整理に必要なツール `process_jam_log` と `compiler_status` をコンパイルしておかなければならない．これらは MSVC ではコンパイルすることができない．

もちろん試験のみなら MSVC だけでも実施可能である．次に実際の退行試験のコマンドを示す．

```
d:
cd ¥boost-1.30.2¥status
..¥bjam --dump-tests test>bjam.log 2>&1
```

以上を実行すると `bjam.log` に退行試験の結果が保存される．次に試験結果を整理する．操作は以下のとおりである．

```
process_jam_log < bjam.log
compiler_status%BOOST_ROOT%cs-win32.html
```

これで本来なら `bjam.log` を整理して各コンパイラごとの正誤表を作ってくれる．

テストごとの試験結果は `http://boost.sourceforge.net/regression-logs/cs-win32.html` を参照してほしい．退行試験の結果をまとめたものを表2に示す．

〔表2〕Boost の各コンパイラでの差異

ライブラリ名	Borland C++	Code Warrior 8.3	GCC 3.3	Intel C++ 7.00	Microsoft Visual C++		
					6.0	7.0	7.1
any	○	○	○	○	○	○	○
array	△	○	○	○	△	△	○
bind	○	○	○	○	○	○	○
concept_check	△	△	○	○	△	△	○
config	△	△	○	△	△	△	○
conversion	○	○	○	○	○	○	○
crc	○	○	○	○	○	○	○
date_time	△	△	○	△	○	○	○
disjoint_sets	×	○	○	○	○	○	○
dynamic_bitset	×	○	○	○	○	○	○
filesystem	△	○	○	○	○	○	○
format	×	×	○	×	×	○	○
function	△	△	○	○	△	△	○
graph	○	○	○	○	○	○	○
integer	△	○	○	○	○	○	○
io	△	○	○	×	×	○	○
iterator	△	△	△	△	△	△	△
lambda	×	○	○	○	×	△	○
math	×	○	△	△	△	△	△
multi_array	△	○	○	△	△	△	○
numeric/interval	△	○	○	○	△	△	○
numeric/ublas	○	△	○	△	○	△	○
optional	○	○	○	○	△	△	○
pool	×	○	○	○	×	×	○
preprocessor	○	○	○	○	○	○	○
property_map	○	○	○	○	○	○	○
random	△	△	○	△	△	△	○
rational	○	○	○	○	○	○	○
regex	△	△	△	△	△	○	○
signals	△	△	○	×	△	△	○
smart_ptr	○	○	○	○	○	△	○
static_assert	△	△	○	○	○	○	○
test	△	△	△	△	△	△	△
thread	×	○	△	○	○	○	○
timer	○	○	○	○	○	○	○
tokenizer	○	○	○	○	○	○	○
tuple	×	○	○	△	○	○	○
type_traits	△	△	△	△	△	△	○
utility	△	○	△	○	○	△	○
variant	△	△	○	△	△	△	○

○＝正常　△＝一部異常　×＝異常

これから標準化に向けて整理されると思われるが，Boost のすべての機能が使えるコンパイラは今のところ存在しない．また，必ずしも退行試験と実際の利用は一致しない．このへんが現時点で Boost の利用をためらう最大の原因である．

また，退行試験の表は常に更新されているので，実際に使う前に調査したほうがよいだろう．

Boost の機能

ここではカテゴリ別に Boost の機能を紹介し，いくつかのコード例を示す．その他の機能は Boost.org に膨大なサンプルコードがあるので，それを参照されたい．なお，Boost.org のカテゴリ分類は機能が重複して記述されているので，本章では適切にまとめてある．

すべてのサンプルコードは MSVC6および MSVC7.1で動作を確認している．Boost ライブラリは共通サーバ `¥¥ad0¥bin¥Boost` にコピーされている．

文字列処理

● conversion

多態キャストとデータ変換機能を実装する．多態キャストと

〔リスト 1〕lexical_castの使用例(TTlexicalcast.cpp)

```
//---------------------------------------------------------
//Boost::lexical_cast:型変換
//---------------------------------------------------------
#include<boost/lexical_cast.hpp>
//#include<string>
#include<iostream>
//#include<strstream>
void TTlexicalcast(void)
{
    using namespace std;
    using namespace boost;
    cout<< "TTlexicalcast:=============================¥n";
   try{
        //--- (1)数値から数字への変換 ---
        int iDt1=1234;
        string st1=lexical_cast<string>(iDt1);
        int iDt2=lexical_cast<int>(st1);
        //--- (2)doubleから文字列への変換 ---
        double dbDt1=1.234;
        string st2=lexical_cast<string>(dbDt1);
        cout<< "String="   <<st1
             << " Int="    <<iDt2
             << " Double=" <<st2
             <<endl;
        //--- (3)エラーが起こる
        st1= "OUK";
        dbDt1=lexical_cast<double>(st1);
    }
    catch(bad_lexical_cast&err) {
        cout<< "Error:" <<err.what() <<endl;
    }
}
```

〔リスト 2〕formatの使用例(TTformat.cpp)

```
//------------------------------------------------------------
//Boost::format:書式文字列生成
//------------------------------------------------------------
#pragma warning(disable : 4786)
#include<iostream>
#include<boost/format.hpp>
void TTformat(void)
{
    using namespace std;
    using namespace boost;
    using namespace boost::io;
    cout<< "TTformat:=================================¥n";
    try{
        string st1;
        //--- (1)sprintfと同じ
        st1=str(format("Numeric=%d Stirng=%s¥n") %30%
                "ABCD表示文字");
        cout<<st1;
        //--- (2)拡張書式
        st1=str(format("No1=%1%No.2=%2%No.3=%3%¥n") % "Type" %
                "safe" % "printf");
        cout<<st1;
        //--- (3)ERRORの例
        st1=str(format("No1=%s No.2=%s¥n") % "OUK");
    }
    catch(too_few_args&err) {
        cout<< "Error:" <<err.what() <<endl;
    }
}
```

はC++のstatic_castにエラー検出機能を加えたキャストである．

データ変換はlexical_castと呼ばれ，intやdoubleの数値を文字列に変換してくれる．使用例をリスト1に示す．

lexical_castではエラーが起これば例外が発生し，try/catchにて例外処理が可能となる．リスト1の(1)はintと文字列の相互変換を示している．リスト1の(2)ではdoubleと文字列の変換例を示す．(3)の変換はdoubleの数値を文字列にて変換したためにエラーが起こることを示している．このとき，catch()のコードが実行される．

● format

型を保証したsprintf()風の書式化を提供する．format()に対して引き数を列挙するにはカンマではなく%演算子を使用する．その他は，いくつかの拡張はあるがsprintf()と同じと考えてよい．リスト2にformatの使用例を示す．

ところで，リスト2のコードをそのままコンパイルすると，MSVC6では警告の山となる．そこで#pragma warningを使って"Warning C4786"を禁止しておく．STLでもそうだが，MSVCではテンプレートのコンパイルによく警告を出す．

リスト2の(2)にあるformat()は，拡張書式"%n%"を示している．これは単純に順次引き数を割り当てる書式である．(3)では引き数の個数が書式と異なるので，例外が起こることを示している．

● regex

正規表現を扱う．この機能を使うためにはリンクライブラリが必要になる．いくつかの機能があり，リスト3に使用例を示す．

一番よく使うのは文字列から正規表現に一致する部分を引き出すregex_search()だろう．これは複数の正規表現を()で続けることにより分割して受け取ることができる．結果はstr(n)に入る．str(0)は全部の条件にマッチした文字列が入る．str(1)は最初の()中の正規表現に一致した文字列で，str(2)は2番目の正規表現一致が代入される．リスト3の(1)の例ではURLが順次分解して代入される．

ただし，regexではシフトJISコードは使えない．リスト3の(2)に示すようにシフトJISで検索すると"A.*"が"ア"に一致してしまう．漢字を扱うときはリスト3の(3)のようにUNICODEを使う．ただし，正規表現側での漢字，たとえば"ア.*"のような表現はうまく一致しない．

加えてregexではgrepと同じ機能をもっている．リスト3の(4)に示すようにgrepに正規表現を渡すと，任意のクラスが呼び出される．関数コール演算子を定義しておくことにより，一致するごとに決まった処理を記述できる．

● spirit

LL構文解析法の骨格で，yacc/lexのような字句解析を扱う．XMLで有名なBNF記法にて構文を定義する．

● tokenizer

文字列を一連のトークンに分割し，イテレータにて読み出せる．

コンテナ

● array

固定長の配列を提供する．STLでは配列を扱うのにvector

〔リスト 3〕regex の使用例(TTregex.cpp)

```
//----------------------------------------------------------------
//Boost::regex:正規表現
//----------------------------------------------------------------
#if 1300<=_MSC_VER
#ifdef _DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥regex¥¥build¥¥vc7¥¥boost_regex_vc7_sssd.lib")
#else   //!_DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥regex¥¥build¥¥vc7¥¥boost_regex_vc7_sss.lib")
#endif  //_DEBUG
#pragma warning(disable : 4244)
#else   //MSCC6
#ifdef _DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥regex¥¥build¥¥vc6¥¥boost_regex_vc6_sssd.lib")
#else   //!_DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥regex¥¥build¥¥vc6¥¥boost_regex_vc6_sss.lib")
#endif  //_DEBUG
#endif  //_MSC_VER
#include<iostream>
#include<boost/regex.hpp>
using namespace std;
using namespace boost;
class IndexClassPred
{
 public:
    //++grepのための関数コール演算子定義
    bool operator()(const match_results<string::const_iterator>&what)
    {
        cout<<what.str(0) <<endl;
        return true;
    }
};
void TTregex(void)
{
    cout<< "TTregex:=================================¥n";
    //--- (1)正規表現で引き出す
    const char*pcS1= "http://www.ouk.jp/kikaku";
    reg_expression<char>rgxS1("([a-z]+:)//(.*)/([a-z]+)");
    match_results<const char*>mrAns1;
    regex_search(pcS1,mrAns1,rgxS1);
    cout<< "(1)"
         <<  "0:" <<mrAns1.str(0)   //全体に一致
         << "1:" <<mrAns1.str(1)    //([a-z]+:)に一致
         << "2:" <<mrAns1.str(2)    //(.*)に一致
         << "3:" <<mrAns1.str(3)    //([a-z]+)に一致
         <<endl;
    //--- (2)SHIFT JISは誤動作する
    const char*pcS2= "アイウエオABCD";
    reg_expression<char>rgxS2("A.*");
    match_results<const char*>mrAns2;
    regex_search(pcS2,mrAns2,rgxS2);
    cout<< "(2)" <<mrAns2.str(0) <<endl;
    //--- (3)漢字はUNICODEにする
    const wchar_t*pwS2=L"アイウエオABCD";
    reg_expression<wchar_t>wrgxS2(L"A.*");
    match_results<const wchar_t*>wmrAns2;
    regex_search(pwS2,wmrAns2,wrgxS2);
    wcout<<L"(3)" <<wmrAns2.str(0) <<endl;
    //---- (4)grepの例
    const string stS  = "The Boost looks like STL oop";
    reg_expression<char>rgxS3(".oo.");
    regex_grep(IndexClassPred(),stS.begin(),stS.end(),rgxS3);
}
```

を用意しているが，動的な配列を目的としている．

● dynamic_bitset

ビット操作を扱う．STLのbitsetの可変サイズ版であり，ほとんど同じインターフェースを持つ．

● multi_array

boost::arrayの多次元版である．STLコンテナに準拠している．STLの多次元配列はvectorをネストして実現できるが，メモリのオーバヘッドが多くなる．

multi_arrayは連続したメモリデータなので，関数に渡しても次元情報が失われることはない．

● property map

一般的なオブジェクトに属性を与える．オブジェクトにキーをマップすることにより実現する．読み書き属性はあらかじめ定義されいる．もちろん独自の属性も扱える．

イテレータ

STLと同じ概念で，datetime，graph，operators，

tokenizer などにもイテレータを含んでいる．

● iterator adaptors

STL のイテレータを拡張する．増加減少関数を定義しておけば，++ や -- の操作が可能なイテレータを自動的に生成してくれる．

関数オブジェクト

STL の関数オブジェクトは mem_fun_ref と bind2nd に代表されるよう多くの制約があった．参照への参照が生成され，コンパイルされないことがある．Boost 関数オブジェクトはそういった STL の問題をほとんど解決している．

● bind と mem_fun

関数オブジェクトに対する操作の集合である．bind は STL の bind1st と bind2nd を改良し，より汎用化したテンプレートである．同様に mem_fun は STL の mem_fun と mem_fun_ref を汎用化したものである．

リスト 4に mem_fun() の使用例を示す．

これらの関数オブジェクト関連の機能は STL と同じ名前なので namespace を注意しなければならない．リスト 4では std を宣言して，boost は明示的に記述している．このため，boost::mem_fun() を std::mem_fun() に書き換えるとコンパイルできなくなる．

〔リスト 4〕mem_fun の使用例（TTmemfun.cpp）

```
//------------------------------------------------------------
//Boost::functional:関数オブジェクトアダプタ
//------------------------------------------------------------
#include<boost/functional.hpp>
#include<algorithm>
#include<iostream>
#include<iterator>
#include<string>
#include<vector>
using namespace std;
class Color
{
    string col;
public:
   Color(const char*cn) : col(cn) {}
   const string&getColor(void)const{return col; }
};
void TTmemfun(void)
{
    cout<< "TTmemfun:================================¥n";
    Color red("Red");
    Color green("Green");
    Color blue("Blue");
    vector<Color*>allc;
    allc.push_back(&red);
    allc.push_back(&green);
    allc.push_back(&blue);
    transform(allc.begin(),allc.end(),
        ostream_iterator<string>(cout, " "),
        boost::mem_fun(&Color::getColor)
    );
    cout<<endl;
}
```

〔リスト 5〕BOOST_PP_REPEAT の記述例

```
#define macro(d,n)d[n],
BOOST_PP_REPEAT(5,macro,d)
```

〔リスト 6〕BOOST_PP_REPEAT の展開

```
d[0],d[1],d[2],d[3],d[4],
```

● function

関数ポインタを関数オブジェクト同様に扱うためのテンプレートである．関数オブジェクトを返す STL 機能の返り値を保存することができる．

● functional

std::functional の改良版で，ほとんど同じ使用法だが，参照引き数でもコンパイルエラーは起こらない．

● lambda

λ抽象の型を実装したテンプレートライブラリである．いくつかの定義により，STL のアルゴリズムに無名の関数オブジェクトの定義方法を提供する．

● ref

参照を汎用関数に渡すためのテンプレートである．

● signals

開発環境 Qt に実装されている“シグナル&スロット”を提供する．シグナルは複数のオブジェクトに対するコールバックを表し，スロットは受け口である．双方とも複数の接続を許可している．

基本テンプレート

● mpl

基本テンプレートのフレームワークを提供する．

● static_assert

静的アサート BOOST_STATIC_ASSERT(x) を提供する．コンパイル時に assert マクロと同じ働きをし，x が偽ならコンパイルエラーとなる．

● call_traits

関数の引き数に適切な型を定義する．

● preprocessor

プリプロセッサ用に繰り返しと再帰を定義する．

もっともよく使うマクロ反復の使用例をリスト 5に示す．この結果，リスト 5はリスト 6のように展開される．

● type_traits

ある型の基本的な特性を識別するためのテンプレートである．たとえば，is_integral<T> は T が整数型がどうか判断する．また，alignment_of<T> は T が必要とするアライメントを返す．

数学

● math

数学に関連したテンプレートの集合である．atanh() や sinc()，sinhc() などの特殊な数学関数や，四元数，八元数を提供する．最大公約数と最小公倍数を扱うクラスも用意され

〔リスト7〕intervalの資料例(TTinterval.cpp)

```
//-----------------------------------------------------------
//Boost::intarval:区間計算
//-----------------------------------------------------------
#include<iostream>
#include<iomanip>
#include<boost/numeric/interval.hpp>
void TTinterval(void)
{
    using namespace std;
    using namespace boost::numeric;
    cout<< "TTinterval:================================¥n";
    // (1)区間 + 区間
    interval<int>rg1(10,100);
    interval<int>rg2(50,150);
    interval<int>rg3=rg1+rg2;
    cout<< "(1)rg1+rg2="
         << "[" <<rg3.lower()
         << "," <<rg3.upper()
         << "]" <<endl;
    // (2)区間 AND 区間
    interval<int>rg4=intersect(rg1,rg2);
    cout<< "(2)rg1&rg2="
         << "[" <<rg4.lower()
         << "," <<rg4.upper()
         << "]" <<endl;
    // (3)区間の検査
    bool fInc1=in(12, rg1);
    bool fInc2=in(102,rg1);
    cout<< "(3)rg1(12)=" <<boolalpha<<fInc1
         << "  rg1(102)=" <<boolalpha<<fInc2
         <<endl;
}
```

ている．

● interval

数学的な"間隔"もしくは"範囲"を提供するテンプレートである．たとえば，aとbの間に存在する数を表す[a,b]と，cからdまでの数を表す[c,d]との四則演算が可能である．リスト7に区間計算の例を示す．リスト7の(1)では区間同士の足し算をし，リスト7の(2)では区間の共通部分を求めている．リスト7の(3)は値が区間に含まれているかどうかの検査をしている．

リスト7のコードは残念ながらMSVC6ではコンパイルできない．退行試験の結果と一致する．後々改善されるだろう．

● integer

汎用的な整数型を簡単に扱うためのテンプレートである．標準C++(1998)のintegerはあいまいな面が多いが，boost::integerは整数の配意を明確に定義できる．

たとえば，int_t<29>は29ビットの符号付き整数である．

● operators

基本的な演算子を定義するのに用いる．

たとえば演算子==と>のオーバロード関数を定義していれば，演算子>=を自動的に定義してくれる．リスト8に簡単なクラス定義の例を示す．

リスト8の定義にて，>，<=，>=などの演算子も使える．

● random

いろいろな方法で乱数を生成する．C言語のrand()はあらゆる精度の点で問題が多い．Boostのrandomはかなり高品位な乱数を発生させることができる．いくつかの乱数発生器といくつかの分布生成器を組み合わせて使う．リスト9にrandomの使用例を示す．

〔リスト8〕operatorsの使用例

```
class MyInt : boost::operators<MyInt>
{
   bool operator<(const MyInt&x)const;
   bool operator==(const MyInt&x)const;
   MyInt&operator+=(const MyInt&x);
   MyInt&operator-=(const MyInt&x);
   MyInt&operator*=(const MyInt&x);
   MyInt&operator/=(const MyInt&x);
   MyInt&operator%=(const MyInt&x);
   MyInt&operator|=(const MyInt&x);
   MyInt&operator&=(const MyInt&x);
   MyInt&operator^=(const MyInt&x);
   MyInt&operator++();
   MyInt&operator--();
};
```

〔リスト9〕randomの使用例(TTrand.cpp)

```
//----------------------------------------------------------------
//Boost::random:乱数発生
//----------------------------------------------------------------
#if 1300<=_MSC_VER
#pragma warning(disable : 4244)
#endif
#define BOOST_NO_LIMITS_COMPILE_TIME_CONSTANTS
#include<iostream>
#include<boost/random.hpp>
#include<time.h>
void TTrand(void)
{
    using namespace std;
    using namespace boost;
    cout<< "TTrand:=================================¥n";
    //--- (1)正規分布による乱数発生
    mt19937 randG1;                                  //疑似乱数
    randG1.seed(static_cast<unsigned int>(time(0)));
    normal_distribution<mt19937>randM1(randG1); //正規分布
    for(int r=0; r<10; r++) {
        cout<<randM1() <<endl;
    }
    //--- (2)整数分布による乱数発生
    cout<< "---------------¥n";
    lagged_fibonacci607 randG2(static_cast<unsigned int>
                              (time(0))); //疑似乱数
    //30～40整数分布
    uniform_int<lagged_fibonacci607>randM2(randG2,30,40);
    for(int n=0; n<10; n++) {
        cout<<randM2() <<endl;
    }
}
```

の使用例を示す．

リスト9の(1)の例ではmt19937乱数発生を正規分布にて取得する．リスト9の(2)ではlagged_fibonacci607乱数発生器を整数分布で取得している．C言語のrand()を整数化するよりは格段に精度が高い．

● rational

有理数すなわち分数を扱うテンプレートである．

● uBLAS

密行列や疎行列などの線形代数を扱う．

正当性とテスト

● concept check

テンプレートの引き数に抽象データ型を利用するためのテンプレート群である．以下の手段を提供する．

1) テンプレート引き数をコンパイル時に検査

2) 要求事項を記述する枠組み
3) テンプレートが要求事項を満足しているかの検査機構
4) STLの要求事項に対応する検査と原型

● test

実行時のコード試験機構を提供している．

データ構造

● any

異なる型の値を格納できる安全で汎用的なコンテナを提供する．Visual Basicの`VARIANT`型とよく似てる．

● compressed_pair

2個の値を返す`std::pair`の拡張版で，クラスの空メンバを最適化する．

● tuple

LISPなどの言語に組み込まれている`TUPLE`を提供するテンプレートである．`TUPLE`はC++言語では使えないが，`boost::tuple`を利用すると，たとえば，複数の値を返す関数の簡単な定義などが便利になる．

メモリ操作

● pool

高速なメモリ割り当てを提供する．一度に小さなオブジェクトを大量に割り当てるときなど効率がよくなる．

〔リスト10〕shared_ptrの使用例(`TTsmartptr.cpp`)

```
//---------------------------------------------------------------
//Boost::smart_ptr:スマートポインタ
//---------------------------------------------------------------
#include<iostream>
#include<boost/shared_ptr.hpp>
using namespace std;
//++ 適当なクラスの定義
class testC
{
 public:
    testC(void) {cout<< "New testC¥n"; }
    virtual~testC() {cout<< "Delete testC¥n"; }
    void print(int i) {cout<< "OUK" <<i<<endl; }
};
//++ (1)スマートポインタ型の定義
typedef boost::shared_ptr<testC>ptrTestC;
//++ ポインタ引き数を持つ関数
static void sub1(ptrTestC pt)
{
    pt->print(1);
}
//++ (2)オブジェクトを生成する関数
static void sub2(void)
{
    ptrTestC ptOrg(new testC);
    ptOrg->print(2);
    sub1(ptOrg);
}
//++ (3)試験プログラムメイン
void TTsmartptr(void)
{
    cout<< "TTsmartptr:==================================¥n";
    sub2();
    cout<< "TTsmartptr::end¥n";
}
```

`malloc()`と同じような使い方をするが，デストラクタに`free()`が含まれている．

● smart_ptr

メモリの自己管理機構を持ち，生成や破棄操作を内部に一般化したテンプレートである．

同じようなテンプレートにSTLの`auto_ptr`があるが，不完全な面が多い．たとえば，`auto_ptr`をコピーして用いると，元の`auto_ptr`がスコープから外れるときに`delete`されるので，両方ともが使用不能になる．ポインタを引き数に使うと，案外コピーという操作は起こりやすい．

しかも`std::auto_ptr`はすでに標準から外されているので，移植性を重視するなら`boost::shared_ptr`を使うべきだ．

`shared_ptr`は内部に参照カウントを持っていて，ポインタのコピーがいくつあるのかを厳密に管理している．参照カウンタが0になったとき`delete`が実行される．

リスト10に`smart_ptr`に含まれる`shared_ptr`の使用例を示す．

リスト10の(1)の`sub2()`でオブジェクトを生成し，コピーポインタをリスト10の(2)の`sub1()`に渡す．ここではスコープから外れるが破棄されない．`sub1()`では正常にポインタとして使え，`sub1()`から抜けるときに破棄される．

● utility

メモリ操作関係のテンプレートの集まりである．

その他

● base-from-member

基底クラスの構築子が呼び出される前に，メンバを初期化するためのテンプレートである．

● crc

CRCを生成するテンプレートを提供する．CRCは2の剰余を使った多項式演算で生成される．多項式のパラメータはテンプレートの引き数で指定できる．

● date_time

日付や時間に関する機能を提供する．日の計算や，時間の計算，日と時間の変換など便利な機能が含まれている．リスト11に使用例を示す．

リスト11の(1)では今週の日付けを列挙している．かつ，`==`演算子によって"本日"を判断している．リスト11の(2)は日付けと時間の演算が，＋や－演算子を使って簡単に実現できることを示している．

● filesystem

OSにより異なるファイル操作を汎用化するためのテンプレートで，以下の機能を汎用化している．

1) パスやファイル名
2) ファイルやディレクトリ操作
3) STLの`fstream`と同様な機能

Information ── エクセルソフト，Rogue Wave Software社の「Stingrey Studio」ら3製品を販売
エクセルソフト(株)は，これまでの(株)ローグウェーブソフトウェアジャパンに代わり，米Rogue Wave Software社のGUIコンポーネント「Stingrey Studio 2003」,「Objection Grid」,「Objective Toolkit」の3製品の国内販売代理店になった．

〔リスト 11〕date_time の使用例（TTdatetime.cpp）

```
//--------------------------------------------------------------
//Boost::date_time:日付時間
//--------------------------------------------------------------
#if 1300<=_MSC_VER
#pragma warning(disable : 4244)
#endif
#ifdef _DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥date_time¥¥
              build¥¥bin¥¥libboost_date_time.lib¥¥msvc¥¥debug¥¥
                  runtime-link-static¥¥libboost_date_time.lib")
#else   //!_DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥date_time¥¥
            build¥¥bin¥¥libboost_date_time.lib¥¥msvc¥¥release¥¥
                  runtime-link-static¥¥libboost_date_time.lib")
#endif  //_DEBUG
#include<iostream>
#include<boost/date_time/gregorian/gregorian.hpp>
#include<boost/date_time/posix_time/posix_time.hpp>
void TTdateTime(void)
{
    using namespace std;
    using namespace boost::gregorian;
    cout<< "TTdateTime:=================================¥n";
    //--- (1)今週の日付けを求める
    date d1=day_clock::local_day();
    date_duration wdN(d1.day_of_week());
    date dWS=d1-wdN;
    date dWE=dWS+date_duration(6);
    for(day_iterator itrD(dWS); itrD<=dWE; ++itrD) {
        if(d1== *itrD)
            cout<< "*";
        else
            cout<< " ";
        cout<<to_iso_extended_string(*itrD)
             << " ("
             <<itrD->day_of_week().as_long_string()
             << ")"
             <<endl;
    }
    //--- (2)時間計算
    using namespace boost::posix_time;
    date d2(2003,Nov,10);
    time_duration tdX(hours(5) +minutes(4) +seconds(3));
    ptime t1(d2,tdX);
    ptime t2=t1-tdX;
    time_duration tdC=t2-t1;
    cout<<to_simple_string(t2)
        << " - "
        <<to_simple_string(t1)
        << " = "
        <<to_simple_string(tdC)
        <<endl;
}
```

〔リスト 12〕filesystem の使用例（TTfilesystem.cpp）

```
//--------------------------------------------------------------
//Boost::filesystem:ファイル操作
//--------------------------------------------------------------
#if 1300<=_MSC_VER
#ifdef _DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥filesystem¥¥
            build¥¥bin¥¥libboost_filesystem.lib¥¥msvc7¥¥debug¥¥
                  runtime-link-static¥¥libboost_filesystem.lib")
#else   //!_DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥filesystem¥¥
          build¥¥bin¥¥libboost_filesystem.lib¥¥msvc7¥¥release¥¥
                  runtime-link-static¥¥libboost_filesystem.lib")
#endif  //_DEBUG
#else   //MSCC6
#ifdef _DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥filesystem¥¥
             build¥¥bin¥¥libboost_filesystem.lib¥¥msvc¥¥debug¥¥
                  runtime-link-static¥¥libboost_filesystem.lib")
#else   //!_DEBUG
#pragma comment(lib, "¥¥¥¥Ad0¥¥bin¥¥Boost¥¥libs¥¥filesystem¥¥
           build¥¥bin¥¥libboost_filesystem.lib¥¥msvc¥¥release¥¥
                  runtime-link-static¥¥libboost_filesystem.lib")
#endif  //_DEBUG
#endif  //_MSC_VER
#include<iostream>
#include<boost/filesystem/operations.hpp>
void TTfilesystem(void)
{
    using namespace std;
    using namespace boost;
    //--- ファイル一覧表示 ---
    cout<< "TTfilesystem:=================================¥n";
    filesystem::directory_iterator itEnd;
    filesystem::directory_iterator itDir(filesystem::
                                          current_path());
    for( ; itDir!=itEnd; itDir++) {
        try{
            if(filesystem::is_directory(*itDir))
                cout<< "<" <<itDir->native_file_string() << ">"
                       <<endl;
            else
                cout<<itDir->leaf() <<endl;
        }
        catch(const std::exception&ex) {
            cout<< "ERROR:" <<itDir->leaf() << ":" <<ex.what()
                  <<endl;
        }
    }
}
```

4) エラー処理

5) 低レベルの便利な機能

リスト 12に filesystem の簡単な例を示す．カレントディレクトリのファイルの一覧を表示する．

MSVC6用の filesytem のライブラリは MSVC7.1では使えない．リンク時にエラーが出る．双方で使えるライブラリセットを作るには MSVC7.1でコンパイルした後，ディレクトリ filesystem¥¥build¥¥bin¥¥libboost_filesystem.lib¥¥msvc を msvc7 に変更する．

退行試験の結果はすべて Pass であるのに，どうしても MSVC6および 7.1では fsteram.hpp がコンパイルできない．いずれ改善されるだろうが，fstream が使えないと実際のファイル入出力を作ることはできない．

● graph

グラフに関する汎用的な機能とアルゴリズムを提供する．アルゴリズムとデータ構造を再利用できるように，グラフの概念そのものを抽象化している．残念ながら膨大すぎてこれ以上の一般的な説明は無理である．

● io state savers

入出力ストリームにて，入出力の状態を保存して戻すための機能を提供する．通常の std::cout はいったん表記書式を変更すると，以降はその表記書式に変わってしまう．

● optional

初期済みのオブジェクトと，そうでないオブジェクトを明確

に区別するためのテンプレートである．ポインタ変数を NULL にすることとよく似ている．

実行時に初期化されているかどうかを検査することにより，未初期化オブジェクトへの読み書きを防ぐことができる．さらに，オブジェクトを未初期化の状態に戻すこともできる．

● timer

時間経過の計測や，時間経過の報告，コードの進行状況を表示する機能を提供している．

● thread

非同期の多重スレッドプログラミングを OS に依存しない形で提供する．

C++ 標準ライブラリでは rand()，strtok()，asctime()，ctime()，gmtime()，localtime() は C から継承して作られているため，スレッド内で呼び出すと問題が起こる．そういった問題も boost::thread では解決されている．

● python

スクリプト言語 Python に C++ のクラスおよび関数をインターフェースする．

おわりに

Boost の機能をひととおり網羅したつもりだが，名前の紹介に終わっているのも多い．消化不良の感は否めないが，どうかお許しいただきたい．

Boost は今も進化を続けており，常に新しい版が提供されている．詳細は Boost.org を参照されたい．なお，本章の日本語の用語については Boost 翻訳プロジェクトに準拠した．

皆様が本章で Boost に興味をもっていただければ幸いである．

参考文献

1) Boost 公式ホームページ，http://www.boost.org/
2) Boost ライブラリドキュメント翻訳プロジェクト，http://www.shibu.jp/wiki/BoostTranslation
3) Effective STL，スコット・メイヤーズ，細谷 昭訳

やの・えつお　yano@ouk.jp　(株)オーク

Chapter 3

汎用的なデータ構造やアルゴリズムを構築するために

標準テンプレートライブラリ STLの概念，そして再考

後藤 正治

本章ではANSI/ISO C++標準の一部であるSTLについて説明する．C++が備えているテンプレートはとても強力な機能で，型などにとらわれることなく汎用的なデータ構造やアルゴリズムを構築することができる．STLはこのテンプレートを用いた汎用的なコンテナ（データ構造）とアルゴリズムのライブラリである．ANSI/ISO C++は1998年に標準として採択され，その一部となったSTLについても一般に広く知られるようになった．

本章ではSTLの利用の裾野を広げるため，その基本概念と簡単な使い方を説明する．そして後半ではSTL利用者の立場から見た実際的な留意点についても述べる．（筆者）

ソフトウェア工学において，ソフトウェアの構成要素を部品化して再利用性を高めることは長年の課題でした．STL以前にもいくつかの方法が提唱されたのですが，部品化を進める代償として実行速度や汎用性が犠牲になることが多く，なかなか決定的な技術は現れませんでした．

そのなかで1994年にHewlett Packard研究所のAlex Stephanov氏によってANSI/ISO C++標準化委員会に紹介されたSTLは，実行速度と汎用性を犠牲にしない画期的なソフトウェア部品化技術として認知され，C++標準ライブラリとしてC++言語仕様の一部になりました．

1994年当時，ANSI/ISO C++は標準化の最終段階に入っていました．STLの追加はそれを大きく遅らせてまで成し遂げるだけの価値があるきわめて意義深いものだったのです．

汎用的なソフトウェア部品のライブラリ

STLはC++の文法を駆使した，たいへん高度なテクニックを使用しています．そのため，初心者には少しわかりにくいかもしれません．

しかし，実現方法の難しさとうらはらに，STLの目指すところは単純です．ソフトウェアをアルゴリズム，コンテナといった構成要素に分解し，それらを自由に組み合せられる汎用部品にしようというのです．

誰でも使える汎用部品にするためには，対象や環境によることなく利用できなくてはなりません．利用できる環境が少なければ利用者が限られたり，いくつかの対立する規格が提唱されたりと，標準化を妨げる状況を誘発してしまいます．

たとえば，考え方のうえだけならばソフトウェアの部品化を目指したライブラリはオブジェクト指向（ポリモーフィズム）によっても実現できます．しかし，オブジェクト指向はSTLに比べて制約が強い技術であるため，統一的な標準を作ることは難しいといえます．実際，オブジェクト指向のライブラリは多数存在しますが，決定的な標準になったものはありません．

また，汎用性があっても従来のプログラミングテクニックに比べて実行速度が劣るようでは利用できる場面が限られてしまい，やはり普及は進みません．

STLで提供されているコンテナやアルゴリズムは，パフォーマンス的にペナルティがない実現方法を追求しています．STLでは，C++のインライン関数というしくみをうまく利用して関数呼び出しのオーバヘッドを回避しています．従来の方法と比べて速度を犠牲にする恐れがないので，安心して使えるのも特徴です．

本章の構成

本章は三つのパートで構成されています．

Part 1では，まず，C++初級および中級者の方を対象に，手続き型プログラミングとの対比において，STLによるソフトウェア部品化の利点を説明します．

Part 2では，まずはSTLを使い始められるよう，簡単なSTLの利用例を示して説明します．

最後にPart 3ではSTL利用に際する実際的考察や注意点について，筆者なりの考えを示します．

今回は，STLについて網羅的に情報を列挙することは避け，逆に部分的な知識でも，とにかく読者がSTLを使い始められるような構成にしました．

また，STLについてはすでに良い書籍が複数入手できるので，STL全体を網羅する情報についてはそちらを参考にしてください．

Part 1　STL によるソフトウェア部品化の考え方と利点

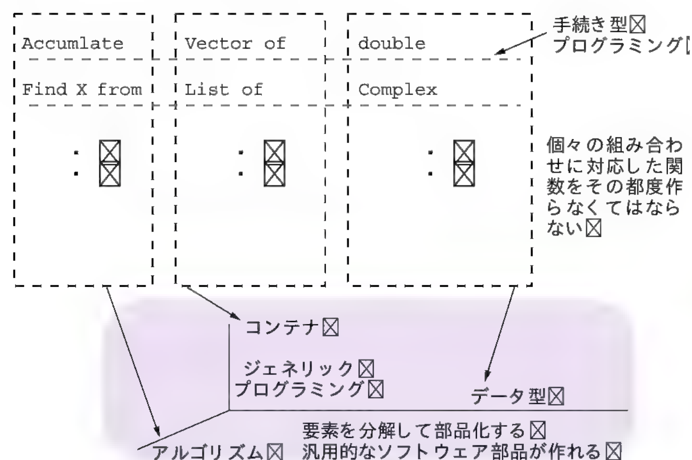
前述したように，STL の目指すところは図 1 のようにソフトウェアを再利用可能な要素に分解し，それを汎用部品ライブラリとして広く提供することです．そのおもなものはコンテナとアルゴリズムです．

そこで，まず，コンテナとはなにか，そしてコンテナとアルゴリズムをどのように分離し，汎用部品化するのかを説明します．Part 1で紹介するサンプルプログラムは考え方を説明するための簡易的なもので，実際のSTLとは異なる点に注意してください．

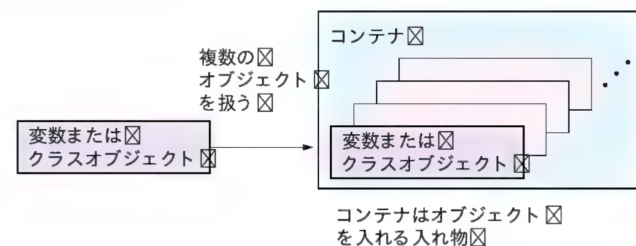
コンテナの一般化とテンプレート

図 2に示すように，コンテナは他のオブジェクトを格納するための入れ物です．多くのプログラミング言語においてもっとも単純なコンテナは配列です．配列とは同一型のデータをメモリ上に複数連続的に格納した集合体で，多数のデータを構造的に扱うことができます．配列は，C++ 言語に組み込まれたしくみですが，それ以外のコンテナはクラスを用いて実現します．そして，そのようなクラスをコンテナクラスと呼びます．

〔図 1〕ソフトウェア要素の分離，部品化



〔図 2〕コンテナの例



● コンテナを使わない悪い例

初心者の中には複合的なデータの集合体を扱うとき，リスト 1 のような書き方をする方もいるかもしれません．じつは筆者もかつてこのような書き方をして，後で痛い目にあったことがあります．

リスト 1では，一見 ClientData というクラスにデータが抽象化されているように見えます．しかし，ClientData クラスでは実は，

{string name,string address,int age}

〔リスト 1〕複合的なデータの集合体を扱う

```
// 悪いデータ構造の作り方
// 個々のデータの概念と集合(コンテナ)の概念が混在している
class ClientsData {
 private:
  string *m_name_ary;
  string *m_address_ary;
  int *m_age_ary;
  int n;
 public:
  ClientsData(int num) {
    n = num;
    m_name_ary = new string[num];
    m_address_ary = new string[num];
    m_age_ary = new int[num];
  }
  void Set(int entry,const string& name,
                          const string& address,int age) {
    m_name_ary[entry] = name;
    m_address_ary[entry] = address;
    m_age_ary[entry] = age;
  }
    ...
  ~ClientsData() {
    delete[] m_name_ary;
    delete[] m_address_ary;
    delete[] m_age_ary;
  }
};
```

〔リスト 2〕コンテナとデータの分離

```
// 改善した例
// 一人分のデータとその集合であるコンテナを分離して定義
class ClientData {
 private:
  string m_name;
  string m_address;
  int m_age;
 public:
  ClientData& operator=(const ClientData& source) {
    *this = source;
    return(*this);
  }
};

template<class T>
class Container {
 private:
    T *m_dat_ary;
 public:
  Container(int num) : n(num) { m_dat_ary = new T[num]; }
  void Set(int entry,const T& source) {
    m_dat_ary[entry] = source;
  }
    ...
  Container() { delete[] m_dat_ary; }
};

typedef ClientsData Container<ClientData>;

// 一歩前進，でも，まだまだ先は長い
```

という一人の顧客に関する情報のまとまりと，それらを集合として扱うという二つの要件が混在して実現されています．これでは，一人一人に関する情報に変化があった場合にも，また，集合としての動作に変化があった場合にも ClientData クラスの変更が必要です．保守性，拡張性，再利用性といった観点から，このような書き方は望ましくありません．

● コンテナとデータの分離

そこで，リスト 2のように一人分のデータを扱う ClientData クラスと，それらの集合体，すなわちコンテナである Container を分離して定義したらどうでしょうか．

ここでは一人分のデータに関する情報と集合としての動作が分離されているため，変更が発生した場合でも影響を局所化できるので好都合です．

コンテナは，STL をはじめとした最新のプログラミングテクニックの中でたいへん重要な位置を占める要素です．コンテナを任意のデータ型に一般化するためには図 3のようにクラステンプレートを使います．

STL では vector，list，deque，set，multiset，map，multimap，stack，queue といったコンテナを汎用的に提供しています．ですから，自分でコンテナを開発する必要はめったになく，ほとんどの場合は STL で提供されているコンテナをそのまま使うか，または，後に述べるアダプタというテクニックを使って，標準コンテナを拡張すれば良いのです．

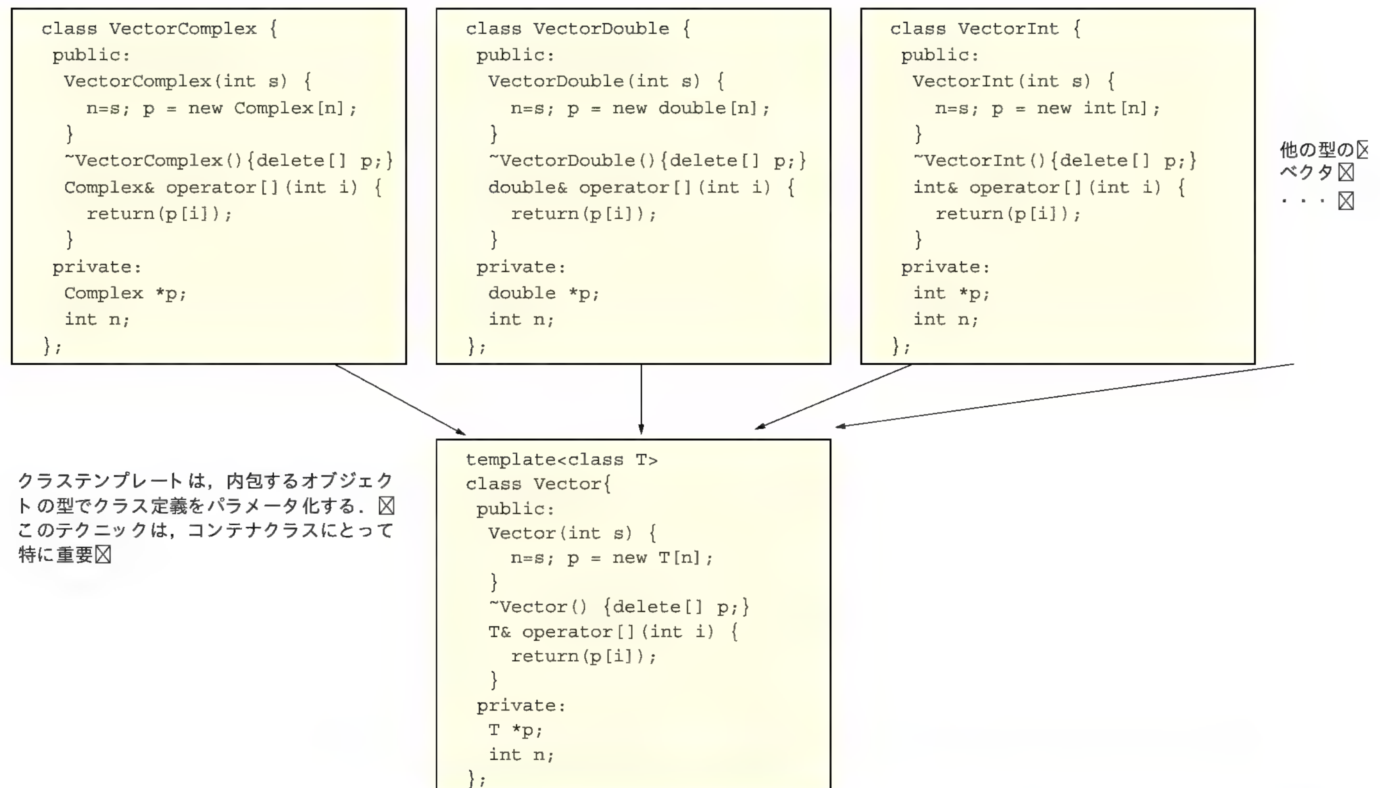
アルゴリズムの汎用化とイテレータ

同種のデータがコンテナのように集合として存在する場合，集合という概念に対して一般的に定義可能なアルゴリズムがあります．たとえば，ある大小関係にしたがって並べ替えたり（sort），特定の条件を満たす要素を探し出したり（search），二つの集合間でコピー，一体化，一致/不一致を判断するといったものです．

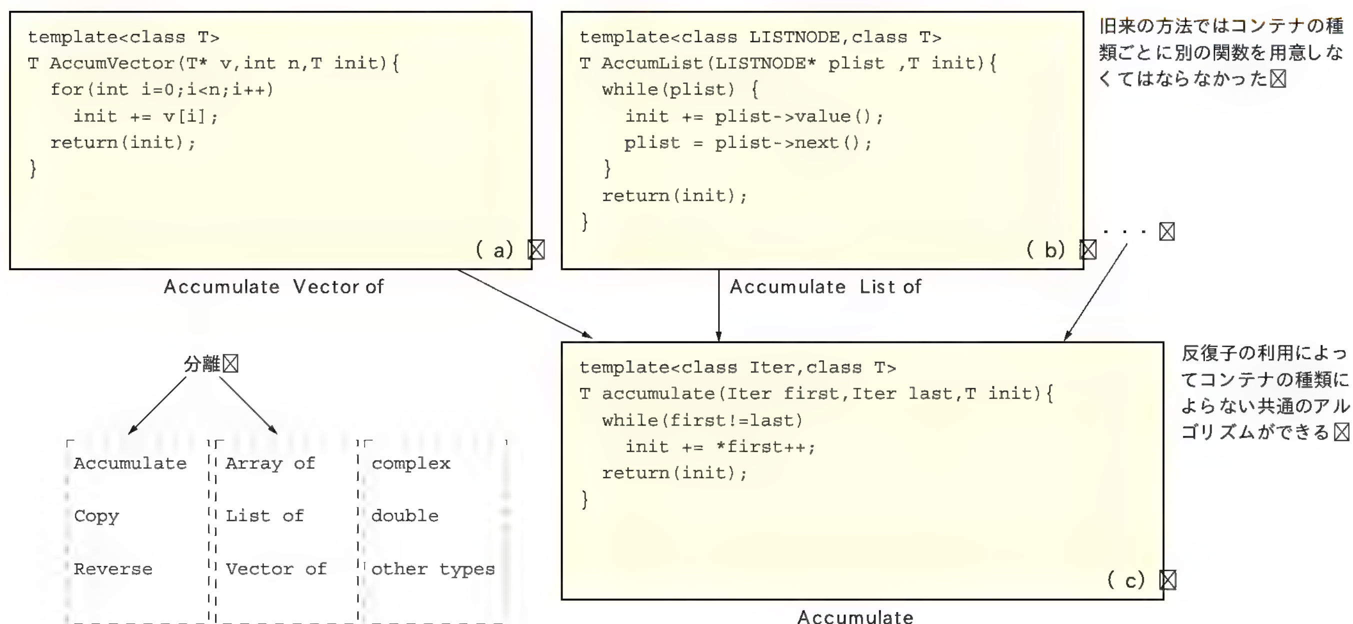
STL では，集合としてのコンテナに対して一般的に定義可能なアルゴリズムを汎用的なライブラリとして提供しています．

汎用的とは，コンテナの種類によらず（ほぼ）同じように共通に利用できるということも意味します．コンテナには配列，vector，list など，さまざまな種類があります．テンプレートによってデータ型の分離には成功しましたが，このままではコンテナの種類ごとにアルゴリズムを開発しなくてはいけません．たとえば，図 4（a）の AccumVector 関数は，配列またはベクタのみに対応しており，リストに対応するものは図 4（b）の AccumList 関数のように別に開発しなくてはいけません．vector と list は特徴が異なるためすべてが同じということはありませんが，AccumVector，AccumList 関数では要素を最初から順番に足し合わせているだけなので，同じ使い方ができても良いはずです．

〔図 3〕クラステンプレート



〔図4〕反復子(イテレータ)によるコンテナとアルゴリズムの分離



〔図5〕イテレータ(反復子)



では，異なるコンテナに対して共通に利用できるアルゴリズムは，果たしてどのように作るのでしょうか．そこで登場するのがイテレータ(反復子)という概念です．

● イテレータ(反復子)

イテレータとはvectorやlistのように要素の集合を扱うデータ構造，つまりコンテナに対して図5のような動作をするものです．

1) 先頭位置の設定
2) 次の要素への移動
3) 要素へのアクセス
4) 走査終了の判定

の四つができればコンテナに対する最低限の操作は可能です．

イテレータの定義方法はいろいろ考えられますが，STLでは配列に対するポインタの表記法を採用しています．ポインタがそのままイテレータとなり得ないコンテナについては，演算子多重定義を使ってポインタのようにふるまうスマートポインタとしてイテレータを定義します．

STLは，イテレータに関する取り決めとして，対象となるコンテナの範囲を，「最初の要素の位置」と「最後+1の要素の位置」で指定します．そしてコンテナの種類に依存しない範囲指定のインターフェースを次のように定義します．

```
template<class Iter>algo(
                        Iter first,Iter last);
```

このイテレータをテンプレート化したインターフェースを使うことによってアルゴリズムライブラリをコンテナにもデータ型にも依存しない汎用的なものにできます．

イテレータを使うもう一つの利点は部分集合の扱いです．(Iter first, Iter last)という方法で範囲を示すため，コンテナの全体のみならず部分についても同じインターフェースで取り扱いができます．

● イテレータカテゴリ

イテレータを使うと多様なコンテナとデータ型に対応できる汎用的なアルゴリズムを書くことができると説明しました．しかし，実際にはコンテナとアルゴリズムの組み合わせには制限

〔表1〕イテレータカテゴリと適用可能演算子，コンテナ，アルゴリズム

イテレータカテゴリ	Random Access Iterator	Bidirectional Iterator	Forward Iterator	Input Iterator	Output Iterator
利用可能な演算子	x= *iter; iter->x; *iter=x; ++iter; --iter; iter==iter2; iter!=iter2; iter[i]; iter+n; iter-n; iter+=n; iter-=n; iter1<iter2 iter1>iter2 iter1<=iter2 iter1>=iter2	x=*iter; iter->x; *iter=x; ++iter; --iter; iter1==iter2; iter1!=iter2;	x=*iter; iter->x; *iter=x; ++iter; iter1==iter2; iter1!=iter2;	x=*iter; iter->x; ++iter; iter1==iter2; iter1!=iter2;	*iter=x; ++iter;
STLコンテナ（左の方が制約が少ない）	配列 vector deque	list set multiset map multimap			
STLアルゴリズム（右のほうが制約が少ない）	randmo_shuffle sort stable_sort partial_sort partial_sort_copy nth_element push_heap pop_heap make_heap sort_heap	copy_backward reverse reverse_copy partition stable_partition inplace_merge next_permutation prev_permutation	find_end find_first_of find_first_of adjacent_find search search_n swap_ranges iter_swap replace replace_if fill generate remove remove_if unique rotate rotate_copy lower_bound upper_bound equal_range binary_search min_element max element	for_each find find_if count count_if mismatch equal copy transform replace_copy replace_copy_if remove_copy remove_copy_if unique_copy partial_sort_copy merge includes set_union set_intersection set_difference set_symmetric _difference lexicographical _compare	copy transform replace_copy replace_copy_if fill_n generate_n remove_copy remove_copy_if unique_copy merge set_union set_intersection set_difference set_symmetric _difference

があります．

データが連続したメモリ領域に並んでいるコンテナでは，即座に任意の要素にアクセスすることができます．このような特性をランダムアクセス性があるといいます．ランダムアクセスが可能なコンテナでは，配列のように，

```
コンテナオブジェクト名[i]
```

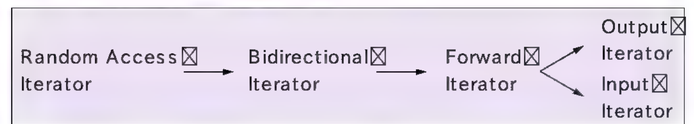
という風に要素にアクセスしたり，

```
イテレータ名 +i
イテレータ名 -i
```

のように足し算や引き算によってイテレータを任意の量だけ移動することもできます．もちろん，インクリメントやデクリメント演算子を使って一つずつ移動することもできます．

このようなイテレータをランダムアクセスイテレータ（Random Access Iterator）と呼びます．

〔図6〕イテレータカテゴリ



STLで提供されるコンテナではvectorとdequeがRandom Access Iteratorを持っています．

一方，前後に隣り合った要素同士はつながりを持っているものの，それ以外の要素間では直接の関係付けができないコンテナもあります．そのようなコンテナでは構造上，要素の移動は前後に一つずつに限定されます．このようなコンテナの反復子はインクリメント，またはデクリメント演算子によって一つずつ移動する能力しかありません．これを双方向イテレータ

(Bidirectional Iterator)といいます．STL で提供されるコンテナでは list，set，multiset，map，multimap が Bidirectional Iterator に対応しています．

さらに一方向にのみつながりを持ち，逆方向の走査ができない構造のコンテナも考えられます．そのようなコンテナでは構造上，要素の移動は順方向に一つずつに限定されます．このようなコンテナのイテレータはインクリメント演算子によって一つずつ順方向に移動する能力しかありません．これを順方向イテレータ(Forward Iterator)といいます．STL では Forward Iterator に分類されるイテレータを持つコンテナは提供されませんが，たとえば単方向リストをコンテナとして実現したとすると，それは順方向イテレータを持つことになります．

〔 リスト 3〕従来的な手法でアルゴリズム開発を行った場合

```
 1: // テンプレートやイテレータといった概念を用いずにアルゴリズムを作成する場合
 2: // [コンテナ][データ型] というコンテナと
 3: // [アルゴリズム][コンテナ][データ型] という関数が多数必要
 4:
 5: class vector_of_int;
 6: class list_of_int;
 7: class deque_of_int;
 8: void sort_vector_of_int(class vector_of_int& v);
 9: void sort_list_of_int(class vector_of_int& v);
10: void sort_deque_of_int(class vector_of_int& v);
11: void binarysearch_vector_of_int(class vector_of_int& v);
12: void binarysearch_list_of_int(class vector_of_int& v);
13: void binarysearch_deque_of_int(class vector_of_int& v);
14: void merge_vector_of_int(class vector_of_int& v1,
                             const class vector_of_int& v2);
15: void merge_list_of_int(class list_of_int& v1,
                           const class list_of_int& v2);
16: void merge_deque_of_int(class deque_of_int& v1,
                            const class deque_of_int& v2);
17:         ...
18: class vector_of_double;
19: class list_of_double;
20: class deque_of_double;
21: void sort_vector_of_double(class vector_of_double& v);
22: void sort_list_of_double(class vector_of_double& v);
23: void sort_deque_of_double(class vector_of_double& v);
24: void binarysearch_vector_of_double(class
                                       vector_of_double& v);
25: void binarysearch_list_of_double(class vector_of_double& v);
26: void binarysearch_deque_of_double(class
                                       vector_of_double& v);
27: void merge_vector_of_double(class vector_of_double& v1,
                                const class vector_of_double& v2);
28: void merge_list_of_double(class list_of_double& v1,
                                const class list_of_double& v2);
29: void merge_deque_of_double(class deque_of_double& v1,
                               const class deque_of_double& v2);
30:         ...
31: class A;
32: class vector_of_A;
33: class list_of_A;
34: class deque_of_A;
35: void sort_vector_of_A(class vector_of_A& v);
36: void sort_list_of_A(class vector_of_A& v);
37: void sort_deque_of_A(class vector_of_A& v);
38: void binarysearch_vector_of_A(class vector_of_A& v);
39: void binarysearch_list_of_A(class vector_of_A& v);
40: void binarysearch_deque_of_A(class vector_of_A& v);
41: void merge_vector_of_A(class vector_of_A& v1,
                           const class vector_of_A& v2);
42: void merge_list_of_A(class list_of_A& v1,
                         const class list_of_A& v2);
43: void merge_deque_of_A(class deque_of_A& v1,
                          const class deque_of_A& v2);
44:         ...
```

この他にも，順方向にしか移動できず，しかも要素への読み込み，または書き込みのいずれかのみが可能な入力イテレータ(Input Iterator)，出力イテレータ(Output Iterator)といった種類があります．

コンテナと反復子に関する制限を体系化した反復子カテゴリを図 6(p.61)と表 1(p.61)に示します．コンテナは右に書いてあるものほど操作の制約が強くなり，アルゴリズムは左に書いてあるものほど適用の制約が強くなります．一部のアルゴリズムで複数のカラムに現れているものがあるのは，それらは複数のコンテナオブジェクトを扱うものだからです(たとえば copy では copy 元が Input Iterator で copy 先が Output Iterator)．

アルゴリズムの引き数がイテレータの場合，そのカテゴリがわかるように名前がつけられています．

Input Iterator
Output Iterator
Forward Iterator
Bidirectional Iterator
Random Access Iterator

ですから，アルゴリズム関数の引き数を見れば，それがどのコンテナと組み合わせて利用できるかがわかります．

コンテナとアルゴリズム一般化の効果

テンプレート，ならびに，イテレータを利用することによって実現したコンテナとアルゴリズムの汎用化は，実際にどのくらい効果があるのでしょうか．リスト 3，リスト 4に，従来の手法でアルゴリズム開発を行った場合と，テンプレートやイテレータといった概念を用いて汎用的なコンテナとアルゴリズムをライブラリ化した STL のインターフェースの場合との比較を示します．

リスト 3ではたくさんの関数が並んでいますが，たとえば 8 行目の sort_vector_of_int は「 int 型」の「 ベクタ」を「 並べ

〔 リスト 4〕テンプレートやイテレータといった概念を用いた場合

```
// テンプレートやイテレータといった概念を用いた STL の考え方
// データ型に依存しない汎用的な[コンテナ]と
// コンテナに依存しない汎用的な[アルゴリズム]のライブラリ

// データ型
class A;
        ...

// コンテナ
template<class T> class vector;
template<class T> class list;
template<class T> class deque;
        ...

// アルゴリズム
void sort(Iterafor first,Iterator last);
void binarysearch(Iterator,Iterator last);
void merge(Iterator,Iterator last,Iterator out);
        ...
```

替える」関数です．次のsort_list_of_intは「int型」の「リスト」を「並べ替える」関数です．少し飛んで43行目のmerge_deque_of_Aは「class A型」の「deque」を統合する関数です．

これらの関数はすべて，ある「データ型」の「コンテナ」に対してある「操作を行う」という観点で作られていることがわかります．あらゆる組み合わせに対応したライブラリを作ろうとすると，

関数の数＝データ型の種類×コンテナの種類
×アルゴリズムの種類

という数の関数が必要になります．

たとえば，5種類のデータ型，3種類のコンテナ，10種類のアルゴリズムに対してなら，5×3×10＝150個の関数が必要です．新しい要素がプログラムに入ってくると，さらに多くの関数が必要になります．

このような方法で作った関数は再利用性が乏しく，限られた状況でしか使うことができません．

一方，リスト4ではコンテナやアルゴリズムが汎用的に定義されているので，インターフェースが単純でコンパクトになっています．必要な関数の数は理論的に，

関数の数＝アルゴリズムの種類

となり，10種類のアルゴリズムに対しては10個の関数を用意すれば良いことになります．

ここまで汎用化が進めば，コンテナやアルゴリズムは標準的なライブラリを提供すれば誰もがそれを利用できるようになるはずです．

以上，STLの基本的な考え方を説明しました．最初に述べたように，ここまでに示したサンプルは説明のための疑似コードで，実際のSTLとは異なります．さて，いよいよPart 2ではSTLを使ってみることにします．

Part 2 最短距離，とりあえずSTLを使ってみる

STLコンテナ

STLでは以下のコンテナが提供されています．

- 列コンテナ　：vector，list，deque
- 連想コンテナ：set，multiset，map，multimap
- その他：string，stack，queue，priority_queue

ここではこれらのうち列コンテナ，ならびに連想コンテナについてサンプルコードを使って一般的な使い方を説明します．ここでは高度な使い方は避け，コンテナを使うのに最低限必要な，

- 宣言
- 要素の追加
- 代入
- 参照

の四つの動作について一般的な使い方を示します．これらが理解できれば，最低限，コンテナを使い始めることができるはずです．

筆者はSTLコンテナの中でもっとも有用で，とりあえずこれだけ使えれば十分といえるコンテナはdeque，list，それにmapだと思います．

以下，各コンテナの説明が続きますが，この三つを中心に使い方をマスターするのも一つの方法です．

ここでは，コンテナにC++基本型のdouble（倍精度実数）を格納しますが，それが他の基本型やクラスオブジェクトであってもコンテナの使い方は同じです．ただし，クラスオブジェクトをコンテナの要素とする場合，そのクラスはpublicなデフォルトコンストラクタ，代入演算子，コピーコンストラクタを持っていることが必要です．また，大小関係が重要になるコンテナ（set，multiset，map，multimap）やアルゴリズムを利用する場合にはoperator==()やoperator<()もpublicメンバとして持っている必要があります．

列コンテナ

● vector

リスト5（p.64）にvectorコンテナの使用例，図7に出力結果を示します．このプログラムは基本的なvectorの使い方を説明する目的で作ったものです．実行は可能ですが，動作に意味はありません．

図8（a）にvectorコンテナの構造の概念図を示します．vectorは可変長配列の機能を持つコンテナです．ランダムアクセスが可能で，末尾において要素を高速に挿入・削除できます．末尾以外の場所での挿入や削除も可能ですが，つねにそれ以降のすべての要素を移動しなくてはならないため，要素数が

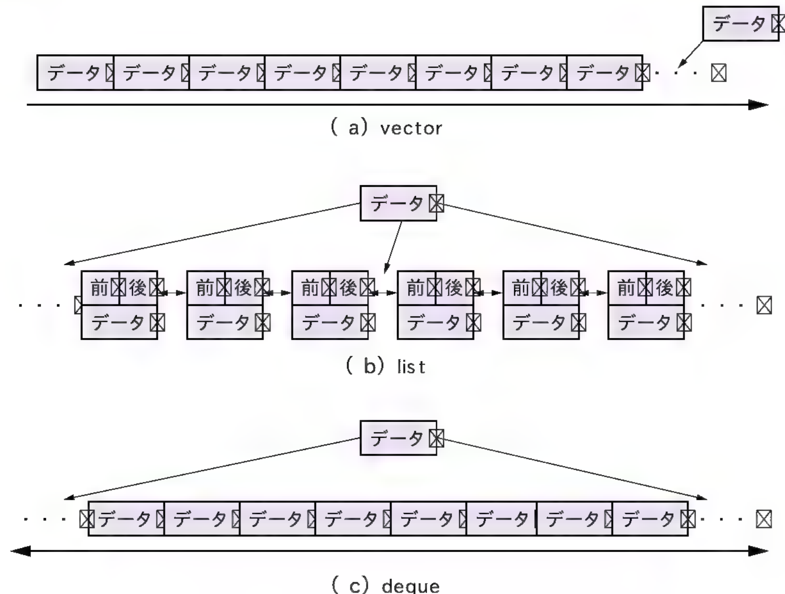
〔図7〕リスト5の出力結果

```
1: 0 1.1 3.14 2.2 3.3 5.5 6.6 7.7 8.8
2: 0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8
3: 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 11.1 11.1 11.1 11.1 11.1
```

〔リスト5〕vectorコンテナの使用例

```
 1: #include <iostream>
 2: #include <vector>
 3: #include <algorithm>
 4: using namespace std;
 5:
 6: /////////////////////////////////////////////////////////
 7: // 関数オブジェクト
 8: void print_value(double x) {
 9:   cout << x << "  " ;
10: }
11:
12: /////////////////////////////////////////////////////////
13: int main() {
14:   int i;
15:
16:   // 宣言 ///////////////////////////////////////////////
17:   vector<double> x;              // 宣言1. size 0 のコンテナ
18:   vector<double> y(10);          // 宣言2. size N のコンテナ
19:   vector<double> z(15,11.1);     // 宣言2. size N のコンテナ
20:   vector<double>::iterator position;
                                     // 宣言3. イテレータの宣言
21:
22:   // 要素の追加 ////////////////////////////////////////
23:   for(i=0;i<10;i++) {
24:     x.push_back(i*1.1);         // 末尾に追加
25:   }
26:
27:   position = find(x.begin(),x.end(),2.2);
28:   if(position!=x.end()) {
29:     x.insert(position,3.14);     // 任意の位置に追加
30:   }
31:
32:   // 要素の削除 ////////////////////////////////////////
33:   if(x.size()>0) x.pop_back();  // 末尾から削除
34:
35:   position = find(x.begin(),x.end(),4.4);
36:   if(position!=x.end()) {
37:     x.erase(position);           // 任意の位置から削除
38:   }
39:
40:   // 代入 //////////////////////////////////////////////
41:   i=0;
42:   for(position=y.begin();position!=y.end();++position) {
43:     *position = (i++)*0.2;       // イテレータを使った代入
44:   }
45:
46:   for(i=0;i<10;i++) {
47:     z[i] = i*0.1;                // operator[] を使った代入
48:   }
49:
50:   // 参照 //////////////////////////////////////////////
51:   for(i=0;i<x.size();i++) {
52:     cout << x[i] << "  " ;       // operator[] を使った参照
53:   }
54:   cout << endl;
55:
56:   for(position=y.begin();position!=y.end();++position) {
57:     cout << *position << "  " ; // イテレータを使った参照
58:   }
59:   cout << endl;
60:
61:   for_each(z.begin(),z.end(),print_value);
                                          // for_eachを使った参照
62:   cout << endl;
63:
64:   return 0;
65: }
66:
67: /////////////////////////////////////////////////////////
```

〔図8〕列コンテナ



(a) vector

(b) list

(c) deque

多くなると時間がかかります．

▶ヘッダファイル

vectorを利用するためには2行目のようにC++標準ヘッダファイル<vector>を読み込む必要があります．3行目の<algorithm>はSTLアルゴリズムを利用するとき必要なヘッダファイルです．リスト5の中で，一部STLアルゴリズムを利用しているのでこれも読み込みます．

▶vectorコンテナの宣言

すべてのSTLライブラリはstdというnamespaceに定義されているため，vectorコンテナを利用するには`std::vector<型名>`'とnamespaceを明記するか，または，4行目のように`using namespace`'命令を使ってstd namespaceを指定したうえで17行目のように`vector<型名>`'と記述します．

じつは，vectorテンプレートには要素の型名の他にもう一つ引き数があり，

```
template<class T,class Allocator=
        std::allocator<T> > class
                        vector{ ... };
```

のように定義されています．

この2番目の引き数はアロケータといってコンテナに必要なメモリ領域の確保を行うものですが，通常はデフォルトで指定されている`std::allocator<T>`のまま使えるので省略することのほうが多いようです．ここではデフォルトの`std::allocator<T>`を利用するため，省略しています．

リスト5の17行〜19行目ではvectorオブジェクトの宣言を行っています．17行目のように引き数なしのデフォルトコンストラクタを使った場合，要素が一つもないコンテナオブジェクトが生成されます．

18行目のように正の整数の引き数を一つ与えると指定した数

だけ要素があらかじめ入ったコンテナオブジェクトができます．各要素は，基本型の場合0，クラスオブジェクトの場合はデフォルトコンストラクタで初期化されます．19行目のように2番目の引き数でコンテナ要素の初期値を指定することもできます．

20行目はvectorコンテナに対応するイテレータの宣言です．イテレータは，たとえ同じvectorコンテナでも格納する要素の型が違えばC++コンパイラは異なる型として扱うので‘vector<型>::iterator’のようにコンテナ型を明示して宣言します．

▶ 要素の追加

リスト5の24行目ではコンテナの末尾に要素を追加するpush_backというメンバ関数を使っています．

29行目ではinsertというメンバ関数を使ってコンテナの途中に要素を追加しています．vectorの場合，挿入位置から最後までの要素を一つずつずらしてコピーしなくてはならないため，コピーする要素数に比例した時間がかかります．挿入位置はイテレータで指定します．ここでは27行目でfindというアルゴリズムを使って要素値が2.2と一致する最初の場所を探しています．一致する要素が見つかった場合，findは有効な位置情報を含むイテレータを返しますが，そうでない場合，無効な位置情報としてx.end()の値を返します．

28行目ではfindの結果が代入されたpositionとx.end()を比較してpositionに有効な位置情報が入っているときのみ挿入動作を行うようチェックしています．

▶ 要素の削除

リスト5の33行目ではコンテナの末尾から要素を削除するpop_backというメンバ関数を使っています．

ここで注意しなくてはらないのは，要素が一つもないコンテナに対してpop_backを使うと，ライブラリによっては異常な動作を引き起こすという点です．STLでは，ライブラリ自身は実行速度を重視し，エラーチェックは利用者が行うという思想になっています．ですから，33行目のようにpop_backを呼ぶ前に，必ず要素が一つ以上あることを確認しなくてはなりません．

36行目では，任意の位置の要素を削除するeraseというメンバ関数を使っています．insert同様，vectorの場合，削除位置から最後までの要素を一つずつずらしてコピーしなくてはならないため，コピーする要素数に比例した時間がかかります．

▶ イテレータを使ったコンテナの走査と要素への代入と参照

42行目，ならびに56行目では，イテレータを使って全要素を走査しています．begin()はコンテナの最初の要素の位置を，end()はコンテナの最後の要素+1の位置を表すイテレータを返すメンバ関数です．42行目や56行目のように，これらをfor文の初期値，終了値として使うと全要素の走査ができます．

43行目ではイテレータを使ってコンテナの要素に代入を行っています．前述したようにイテレータはC/C++のポインタをモデルに定義されているため，単項演算子‘*’を使って参照すると，要素の値の操作が可能です．ここでは‘*position=[値];’という形で代入を行っています．57行目では同様にイテレータを使ってコンテナ要素の値を参照しています．単項演算子‘*’を使って値を操作するやり方は，どのイテレータでも備えている機能です．

▶[ ]演算子を使った要素への代入・参照

47行目では配列のように[]演算子を使って代入を行っています．vectorはランダムアクセスイテレータを持つコンテナなので，このような操作が可能です．

52行目では同じく[]演算子を使って要素の値を参照しています．51行目のfor文の中で使っているsize()は，そのときコンテナに格納されている要素の数を返すメンバ関数なので，ランダムアクセスイテレータをもつコンテナの要素を走査するならば，51行～53行目のような書き方もできます．ただし，できればこのような書き方は避けたほうが賢明です．ランダムアクセスイテレータを持たないコンテナではこのような書き方ができないため，後になってコンテナの種類を変更しにくくなるからです．

もちろん，ランダムアクセスが必要な場合には[]演算子を使えば良いのですが，この例のように順次走査を行っているだけの場合はイテレータを使った方法のほうが一般性があります．

▶for_eachアルゴリズムを使ったコンテナの走査と要素の参照

61行目では，STL初心者の方には見慣れない方法でコンテナの走査を行っています．for_eachアルゴリズムは<algorithm>の中で定義されていますが，その内容はきわめて単純で，第1，第2引き数のイテレータで指定された範囲の要素に対して第3引き数で指定された操作(これを関数オブジェクトと呼ぶ)を実行するものです．ここでは第1，第2引き数にはz.begin()，z.end()が与えられているので，コンテナ全体の走査を行います．第3引き数として与えられたprint_valueはリスト5の8行～10行目に定義されている関数で，for_eachアルゴリズムは走査する要素ごとにこの関数を呼び出します．一つの方法として紹介しましたが，本稿ではPart 3で説明する理由から，for_eachと関数オブジェクトについては詳しい説明を省略します．

● list

リスト6にlistコンテナの使用例を，図9に出力結果を示します．

図8(b)にlistコンテナの構造の概念図を示します．listは双方向リンクリストといって，直前と直後に隣接する要素に高速に移動する双方向イテレータを備えたコンテナです．一つずつ順番に移動するのは前後両方向にできますが，vectorのように[]演算子を使ったランダムアクセスはできません．listではコンテナ中のどの位置でも高速に挿入や削除が可能です．

▶ ヘッダファイル

listを利用するためには2行目のようにC++標準ヘッダ

〔リスト 6〕listコンテナの使用例

```
 1: #include <iostream>
 2: #include <list>
 3: #include <algorithm>
 4: using namespace std;
 5:
 6: ////////////////////////////////////////////////////////////
 7: // 関数オブジェクト
 8: void print_value(double x) {
 9:   cout << x << "  " ;
10: }
11:
12: ////////////////////////////////////////////////////////////
13: int main() {
14:   int i;
15:
16:   // 宣言 ////////////////////////////////////////////////
17:   list<double> x;                 // 宣言1. size 0 のコンテナ
18:   list<double> y(10);             // 宣言2. size N のコンテナ
19:   list<double> z(5,11.1);         // 宣言2. size N のコンテナ
20:   list<double>::iterator position;
                                      // 宣言3. イテレータの宣言
21:
22:   // 要素の追加 ///////////////////////////////////////
23:   for(i=0;i<10;i++) {
24:     x.push_back(i*1.1);          // 末尾に追加
25:   }
26:
27:   for(i=0;i<5;i++) {
28:     z.push_front(i*0.1);         // 先頭に追加
29:   }
30:
31:   position = find(x.begin(),x.end(),2.2);
32:   if(position!=x.end()) {
33:     x.insert(position,100);      // 任意の位置に追加
34:   }
35:
36:   // 要素の削除 //////////////////////////////////////////
37:   x.pop_back();                   // 末尾から削除
38:
39:   x.pop_front();                  // 先頭から削除
40:
41:   position = find(x.begin(),x.end(),4.4);
42:   if(position!=x.end()) {
43:     x.erase(position);            // 任意の位置から削除
44:   }
45:
46:   // 代入 ////////////////////////////////////////////////
47:   i=0;
48:   for(position=y.begin();position!=y.end();++position) {
49:     *position = (i++)*0.2;       // イテレータを使った代入
50:   }
51:
52:   // 参照 ///////////////////////////////////////////////
53:   for(position=x.begin();position!=x.end();++position) {
54:     cout << *position << "  " ; // イテレータを使った参照
55:   }
56:   cout << endl;
57:
58:   for_each(y.begin(),y.end(),print_value);
                                      // for_eachを使った参照
59:   cout << endl;
60:
61:   for_each(z.begin(),z.end(),print_value);
                                      // for_eachを使った参照
62:   cout << endl;
63:
64:   return 0;
65: }
66:
67: ////////////////////////////////////////////////////////////
```

〔図 9〕リスト 6の出力結果

```
1: 1.1 100 2.2 3.3 5.5 6.6 7.7 8.8
2: 0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8
3: 0.4 0.3 0.2 0.1 0 11.1 11.1 11.1 11.1 11.1
```

ファイル <list> を読み込む必要があります．

▶ listコンテナの宣言

listオブジェクトの宣言はvectorと同様に行います．std::list<型名>とnamespaceを明記するか，または4行目のようにusing namespace命令を使ってstd namespaceを指定したうえで17行目のようにlist<型名>と記述します．アロケータがデフォルト引き数として存在しますが，vectorと同様に，通常は省略してデフォルトのまま利用します．

リスト 6の17行目から19行目のlistオブジェクト宣言はvectorの場合とまったく同じです．20行目のlistコンテナに対応するイテレータの宣言も同様です．

▶ 要素の追加

リスト 5の24行目ではコンテナの末尾に要素を追加するpush_backというメンバ関数を使っています．28行目ではpush_frontというメンバ関数を使って，コンテナの先頭に要素を追加しています．これはvectorにはなかった機能です．

29行目ではinsertというメンバ関数を使ってコンテナの途中に要素を追加しています．listでは任意の位置への挿入を高速に行えるので，push_frontのようなメンバ関数が準備されており，また，insertも高速に実行できます．

▶ 要素の削除

リスト 6の37行目でpop_backを使ってコンテナの末尾から要素を削除できるのはvectorと同じです．39行目ではpop_frontメンバ関数を使ってコンテナの先頭から要素を削除しています．これもvectorにはなかった機能です．43行目では任意の位置の要素を削除するeraseというメンバ関数を使っています．listの場合，上記の三つとも高速に実行できます．

▶ イテレータを使ったコンテナの走査と要素への代入・参照

リスト 6の48行～50行目，ならびに53行～56行目でイテレータを使って全要素を走査し，代入・参照を行っている要領は，リスト 5で示したvectorの場合と同様です．

▶ for_eachアルゴリズムを使ったコンテナの走査と要素の参照

61行目のfor_eachアルゴリズムを使ったコンテナの走査はvectorの場合と同様です．

● deque

リスト 7にdequeコンテナの使用例を示します．

図 8（c）（p.64）にdequeコンテナの構造の概念図を示します．dequeはvector同様ランダムアクセスイテレータを持つコンテナです．トランプカードの束のことをデックといいますが，ちょうどその一番上と一番下からカードの出し入れができるようなイメージで作られたコンテナです．vectorのように[]演算子を使ったランダムアクセスができるのとコンテナの先頭，末尾のどちらにも高速に要素の挿入・削除ができる点

〔リスト7〕dequeコンテナの使用例

```
 1: #include <iostream>
 2: #include <deque>
 3: #include <algorithm>
 4: using namespace std;
 5:
 6: //////////////////////////////////////////////////////////
 7: // 関数オブジェクト
 8: void print_value(double x) {
 9:   cout << x << "  " ;
10: }
11:
12: //////////////////////////////////////////////////////////
13: int main() {
14:   int i;
15:
16:   // 宣言 ////////////////////////////////////////////////
17:   deque<double> x;                // 宣言1．size 0 のコンテナ
18:   deque<double> y(10);            // 宣言2．size N のコンテナ
19:   deque<double> z(15,11.1);       // 宣言2．size N のコンテナ
20:   deque<double>::iterator position;
                                  // 宣言3．イテレータの宣言
21:
22:   // 要素の追加 ////////////////////////////////////////
23:   for(i=0;i<10;i++) {
24:     x.push_back(i*1.1);          // 末尾に追加
25:   }
26:
27:   for(i=0;i<5;i++) {
28:     x.push_front(i*0.1);         // 先頭に追加
29:   }
30:
31:   position = find(x.begin(),x.end(),2.2);
32:   if(position!=x.end()) {
33:     x.insert(position,3.14);      // 任意の位置に追加
34:   }
35:
36:   // 要素の削除 ////////////////////////////////////////
37:   x.pop_back();                  // 末尾から削除
38:
39:   x.pop_front();                 // 先頭から削除
40:
41:   position = find(x.begin(),x.end(),4.4);
42:   if(position!=x.end()) {
43:     x.erase(position);           // 任意の位置から削除
44:   }
45:
46:   // 代入 ///////////////////////////////////////////////
47:   i=0;
48:   for(position=y.begin();position!=y.end();++position) {
49:     *position = (i++)*0.2;       // イテレータを使った代入
50:   }
51:
52:   for(i=0;i<10;i++) {
53:     z[i] = i*0.1;                // operator[] を使った代入
54:   }
55:
56:   // 参照 ///////////////////////////////////////////////
57:   for(i=0;i<x.size();i++) {
58:     cout << x[i] << "  " ;       // operator[] を使った参照
59:   }
60:   cout << endl;
61:
62:   for(position=y.begin();position!=y.end();++position) {
63:     cout << *position << "  " ; // イテレータを使った参照
64:   }
65:   cout << endl;
66:
67:   for_each(z.begin(),z.end(),print_value);
                                 // for_eachを使った参照
68:   cout << endl;
69:
70:   return 0;
71: }
72:
73: //////////////////////////////////////////////////////////
```

〔図10〕リスト7の出力結果

```
1: 0.3 0.2 0.1 0 0 1.1 3.14 2.2 3.3 5.5 6.6 7.7 8.8
2: 0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8
3: 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 11.1 11.1 11.1 11.1 11.1
```

が特徴です．

リスト7の全体を眺めると，リスト5で示したvector，リスト6で示したlistの例とほとんど変わりないことがわかります．もう三つ目なので詳しく説明する必要はないでしょう．

このようにコンテナの種類が変わってもいくつかの基本となる操作方法が変わらないのがSTLコンテナの特徴です．

ここでは，主な違いである要素の追加，削除時の実行効率についてのみ簡単に説明します．

▶ 先頭，末尾での追加，削除

24行目，28行目ではそれぞれ末尾，先頭への要素の追加を，37行目，39行目ではそれぞれ末尾，先頭からの要素の削除を行っています．dequeはこのいずれの操作も高速に一定時間で実行できます．

▶ 途中での追加，削除

33行目，43行目では，それぞれコンテナの途中で要素の追加，削除を行っています．dequeでこれを行うと，指定された位置から先頭，または末尾の近い方までの要素を一つずつずらしてコピーします．ですから，実行も先頭，または末尾の近い方までの数に比例した時間がかかります．

連想コンテナ

STLでは連想コンテナとしてset，multiset，map，multimapを準備しています．ここまで説明した列コンテナはコンテナの種類によらず，似たような使い方ができました．これから説明する連想コンテナは，イテレータの使い方など，基本的なところは踏襲しているものの，性質や使い方が異なります．

● set，multiset

リスト8にset，multisetコンテナの使用例を，図11に出力結果を示します．setは要素を常に大小順に並べた状態で保持するコンテナです．大小順の並べ替えは，要素が挿入されるときに行われます．listと同じように双方向イテレータを持っており，前後両方向への走査が可能です．

▶ ヘッダファイル

setとmultisetを利用するためには2行目のようにC++標準ヘッダファイル<set>を読み込む必要があります．この中にはsetとmultiset両方のコンテナが定義されています．

〔リスト 8〕set，multisetコンテナの使用例

```
 1: #include <iostream>
 2: #include <set>
 3: #include <algorithm>
 4: using namespace std;
 5:
 6: ///////////////////////////////////////////////////////////
 7: void print_value(double x) {
 8:   cout << x << "  " ;
 9: }
10:
11: ///////////////////////////////////////////////////////////
12: double randomtable[]
13:   = { 9.2, 1.2, 3.4, 4.5, 3.4, 5.1, 4.5, 0.1, 7.8, 2.4, 0 };
14:
15: ///////////////////////////////////////////////////////////
16: void Set1() {
17:   int i;
18:   cout << "# set" << endl;
19:
20:   // 宣言 ///////////////////////////////////////////////////
21:   set<double> x;                    // 宣言1. size 0 のコンテナ
22:   set<double>::iterator position;  // 宣言2. イテレータの宣言
23:
24:   // 要素の追加 ////////////////////////////////////////
25:   for(i=0;i<10;i++) {
26:     cout << randomtable[i] << "  ";
27:     x.insert(randomtable[i]); // 追加時に大小順にならべられる
28:   }
29:   cout << endl;
30:
31:   // 要素の削除 ////////////////////////////////////////
32:   position = find(x.begin(),x.end(),5.1);
33:   if(position!=x.end()) {
34:     x.erase(position);              // 任意の位置から削除
35:   }
36:
37:   // 代入 //////////////////////////////////////////////
38:   // 一旦挿入された set の要素に代入・変更はできない
39:
40:   // 参照 ///////////////////////////////////////////////////
41:   for(position=x.begin();position!=x.end();++position) {
42:     cout << *position << "  " ;    // イテレータを使った参照
43:   }
44:   cout << endl;
45:
46:   for_each(x.begin(),x.end(),print_value);
                                    // for_each を使った参照
47:   cout << endl;
48: }
49:
50: ///////////////////////////////////////////////////////////
51: void Multiset1() {
52:   int i;
53:   cout << "# multiset" << endl;
54:
55:   // 宣言 ///////////////////////////////////////////////////
56:   multiset<double> x;            // 宣言1. size 0 のコンテナ
57:   multiset<double>::iterator position;
                                 // 宣言2. イテレータの宣言
58:
59:   // 要素の追加 ////////////////////////////////////////
60:   for(i=0;i<10;i++) {
61:     cout << randomtable[i] << "  ";
62:     x.insert(randomtable[i]);   // 追加時に大小順にならべられる
63:   }
64:   cout << endl;
65:
66:   // 要素の削除 ////////////////////////////////////////
67:   position = find(x.begin(),x.end(),5.1);
68:   if(position!=x.end()) {
69:     x.erase(position);          // 任意の位置から削除
70:   }
71:
72:   // 代入 //////////////////////////////////////////////
73:   // 一旦挿入された multiset の要素に代入・変更はできない
74:
75:   // 参照 ///////////////////////////////////////////////////
76:   for(position=x.begin();position!=x.end();++position) {
77:     cout << *position << "  " ; // イテレータを使った参照
78:   }
79:   cout << endl;
80:
81:   for_each(x.begin(),x.end(),print_value);
                                 // for_each を使った参照
82:   cout << endl;
83: }
84:
85: ///////////////////////////////////////////////////////////
86: int main() {
87:   Set1();
88:   Multiset1();
89:   return 0;
90: }
91:
92: ///////////////////////////////////////////////////////////
```

〔図11〕リスト 8の出力結果

```
#set
9.2 1.2 3.4 4.5 3.4 5.1 4.5 0.1 7.8 2.4
0.1 1.2 2.4 3.4 4.5 7.8 9.2
0.1 1.2 2.4 3.4 4.5 7.8 9.2
#multiset
9.2 1.2 3.4 4.5 3.4 5.1 4.5 0.1 7.8 2.4
0.1 1.2 2.4 3.4 3.4 4.5 4.5 7.8 9.2
0.1 1.2 2.4 3.4 3.4 4.5 4.5 7.8 9.2
```

▶要素の追加

リスト 8の27行目でinsertメンバ関数を使ってsetに要素を追加しています．列コンテナでは，insertに挿入位置を示すイテレータも与えましたが，setでは不要です．いずれにしろ大小順に並べ替えられるので，ユーザー側が挿入位置を指定することは無意味だからです．

▶要素の削除

32行目でfindアルゴリズムを使って5.1と一致する要素の位置を示すイテレータを取得し，それを34行目で削除しています．

▶イテレータを使ったコンテナの走査と要素の参照

41行〜47行目では，イテレータを使って全要素を走査しています．このやり方は列コンテナと同じです．

▶setとmultisetの違い

16行〜48行目のsetを使ったプログラムと51行〜83行目までのmultisetを使ったプログラムはまったく同じように書いてあります．setの場合，同じ値の要素を重複して持つことはなく，同じ値が何回も挿入された場合でもその値の要素は一つしか持ちません．一方，multisetの場合は同じ値が複数回挿入された場合，その回数分だけ重複した要素を持ちます．

▶set，multisetの用途

set，multisetは，要素が常に大小順に並んでいるのがとりえのコンテナです．筆者が実験したところでは，すべての要素を集め終わった後でのみ並べ替えが必要ならば，setやmultisetを使うより，dequeコンテナと後述するsortアルゴリズムを使ったほうが高速でした．set，multisetは，挿

〔リスト 9〕map，multimapコンテナの使用例

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>
using namespace std;

/////////////////////////////////////////////////////////
char* names[] = { "def" , "abc" , "mno" , "stu" , "def" ,
                                         "ghi" , 0 };
double vals[] = { 1.23  , 4.56  , 7.89  , 3.01  , 0.12  ,
                                         3.45  , 0 };

/////////////////////////////////////////////////////////
void print_value(const pair<string,double>& x) {
  cout << x.first << "=" << x.second << "  " ;
}

/////////////////////////////////////////////////////////
void Map1() {
  int i;
  cout << "# map" << endl;

  // 宣言 ////////////////////////////////////////////////
  map<string,double> x;          // 宣言1. size 0 のコンテナ
  map<string,double>::iterator position;
                                 // 宣言2. イテレータの宣言
  pair<string,double>  key_value_pair;
                                 // 宣言3. mapの要素の宣言

  // 要素の追加 ////////////////////////////////////////
  i=0;
  while(names[i]) {
    cout << names[i] << "=" << vals[i] << "  ";
    x[ names[i] ] = vals[i];   // operator[]による追加
    ++i;
  }
  cout << endl;

  key_value_pair.first = "stu";
  key_value_pair.second = 9.01;
  x.insert(key_value_pair);    // insert を使った追加

  // 削除 //////////////////////////////////////////////
  x.erase("mno");              // Keyで位置を指定して削除

  // 代入 //////////////////////////////////////////////
  x["def"] = 6.78;   // operator[]を使った代入

  position = x.begin();
  (*position).second = 9.99; // iterator を使った代入(value)

  // 参照 //////////////////////////////////////////////
  cout << "ghi" << "=" << x["ghi"] // operator[]を使った参照
       << " (size=" << x.size() << ")   ";
  cout << "pqr" << "=" << x["pqr"]
                          // operator[]を使った参照(追加)
       << " (size=" << x.size() << ")" << endl;

  for(position=x.begin();position!=x.end();++position) {
    cout << (*position).first << "="
                              // iteratorを使った参照1
         << (*position).second << "  " ;
                              // iteratorを使った参照2
  }
  cout << endl;

  for_each(x.begin(),x.end(),print_value);
                              // for_each を使った参照
  cout << endl;
}

/////////////////////////////////////////////////////////
void Multimap1() {
  int i;
  cout << "# multimap" << endl;

  // 宣言 ////////////////////////////////////////////////
  multimap<string,double> x;     // 宣言1. size 0 のコンテナ
  multimap<string,double>::iterator position;
                                 // 宣言2. イテレータの宣言
  pair<string,double>  key_value_pair;
                                 // 宣言3. multimapの要素の宣言

  // 要素の追加 ////////////////////////////////////////
  i=0;
  while(names[i]){
    cout << names[i] << "=" << vals[i] << "  ";
    key_value_pair.first = names[i];
    key_value_pair.second = vals[i];
    x.insert(key_value_pair);  // insert を使った追加
    ++i;
  }
  cout << endl;

  // 削除 //////////////////////////////////////////////
  x.erase("mno");              // Keyで位置を指定して削除

  // 代入 //////////////////////////////////////////////
  position = x.begin();
  (*position).second = 9.99; // iterator を使った代入(value)

  // 参照 //////////////////////////////////////////////
  for(position=x.begin();position!=x.end();++position) {
    cout << (*position).first << "="
                              // iteratorを使った参照1
         << (*position).second << "  " ;
                              // iteratorを使った参照2
  }
  cout << endl;

  for_each(x.begin(),x.end(),print_value);
                              // for_each を使った参照
   cout << endl;
}

/////////////////////////////////////////////////////////
int main() {
  Map1();
  Multimap1();
  return 0;
}

/////////////////////////////////////////////////////////
```

入が比較的頻繁に行われ，しかも，そのつど要素がきちんと並べ替えられている必要がある場合に使うと良いでしょう．

● map，multimap

リスト 9にmap，multimapコンテナの使用例を，図 12に出力結果を示します．

● map

mapは連想配列という，たいへん有用な機能を提供するコンテナです．連想配列というのは添字として整数以外の任意の型の情報をとることができる配列です．情報を検索するとき，検

〔図 12〕リスト 9の出力結果

```
#map
def=1.23 abc=4.56 mno=7.89 stu=3.01 def=0.12 ghi=3.45
ghi=3.45(size=4)  pqr=0(size=5)
abc=9.99 def=6.78 ghi=3.45 pqr=0 stu=3.01
abc=9.99 def=6.78 ghi=3.45 pqr=0 stu=3.01
#multimap
def=1.23 abc=4.56 mno=7.89 stu=3.01 def=0.12 ghi=3.45
abc=9.99 def=1.23 def=0.12 ghi=3.45 stu=3.01
abc=9.99 def=1.23 def=0.12 ghi=3.45 stu=3.01
```

索のKeyとなる情報はかならずしも何番目の要素というような整数情報ではなく，たとえば名前，職業，電話番号といった順序空間に並んでいない情報ではないでしょうか．そのようなKeyを元に，高速に得たい情報を検索するのが連想配列です．

mapでは検索のKeyになる情報と，それに対応する値の情報がペアになって一つの要素となります．要素がmapに挿入されるときにはsetと同じようにKeyに対して大小順に並べ替えられて格納されます．そのため常に二分検索の原理が利用でき，高速にKeyを検索できます．

mapではKeyの値が等しい要素を複数格納することはできません．また，listと同じように双方向イテレータを持っており，前後両方向への走査が可能です．

▶ヘッダファイル

mapを利用するためには，リスト9の2行目のようにC++標準ヘッダファイル<map>を読み込む必要があります．この中にはmapとmultimap両方のコンテナが定義されています．

▶要素の追加

mapでは他のコンテナとは異なる方法で要素の追加を行います．リスト9の30行目で[]演算子を使って配列のように代入を行っています．ここでは[]演算子に与えられたname[i]がKey，代入されるvals[i]が値の情報となり，それらがペアとなってmapに格納されます．

37行目ではinsertメンバ関数を使って要素を挿入していますが，要素の型は24行目で宣言されているようにpair<string,double>となるので注意が必要です．ここでは説明のために書きましたが，通常はmapに35行～37行目のような書き方をすることはあまりありません．

▶要素の削除

40行目でeraseメンバ関数を使ってKeyが“mno”と一致する要素を削除しています．このようなやり方はmap独特のものです．

▶代入

要素への代入は43行目のように，配列への代入のような記述で行います．じつはmapでは，要素の追加，代入，参照はまったく同じしくみで行われます．[]演算子が使われると，与えられた添字にKeyが一致する要素が検索されます．もし，一致する要素がなければ，そのKeyに対応する要素が自動的に追加されます．そして，その要素の値への参照が返されます．

参照する場合にはそのままその値が使われ，代入の場合には参照（つまりアドレス）を通じて書き込みが行われます．

▶イテレータを使ったコンテナの走査と要素の参照

mapではイテレータによる順序走査ができないと思っている方もいるのではないかと思いますが，じつは可能です．その方法を54行～57行目に示します．

mapコンテナの要素はKeyと値のペアです．イテレータによって走査できる要素はこのペアなので，そこからKeyと値をそれぞれ参照するには55行目，56行目のようにペアのfirst，secondというメンバを参照します．

同様のことは60行目のようにfor_eachアルゴリズムと12行～14行目で定義されている関数オブジェクトを使っても可能です．

▶mapの用途

mapはたいへん用途が広く，有用なコンテナです．辞書，変数テーブルなど，実際のアプリケーションの中での用途が多数あります．mapを使うと，以前は数千行も書かなくてはならなかった高度なアプリケーションを簡潔に書くことができます．ぜひともマスターしたいコンテナです．

● multimap

multimapでは，Keyの値が等しい要素を複数格納することができます．

それ以外の特徴はmapと同じです，といいたいところですがそうはいきません．Keyが等しい要素が複数あるということは[]演算子により一意に要素を特定することができないということです．そのため，multimapには[]演算子は用意されておらず，連想配列としての機能を簡潔に果たすことができません．

リスト9の65行～101行目にmultimapを使ったプログラムを示します．ここまで読み進んでいれば，内容については理解できると思うので詳しい説明は省略します．

コンテナアダプタ

ここまではSTLの基本となる列コンテナと連想コンテナの説明でした．次に，STLコンテナを拡張して作られたコンテナアダプタについて説明します．

プログラムを開発していると既成のSTLコンテナでは機能が足りなかったり，少し違ったインターフェースが欲しくなることがあります．そのようなときに使うと便利なテクニックがデザインパターンとしても紹介されているアダプタです．

アダプタとは，既存のクラスの機能を拡張･変更するために既存クラスを継承するか，またはメンバとして持つテクニックのことです．

アダプタは，

```
template<class T,class Container>
class adapter_by_inheritance : Container{
 public:
   // 追加するインターフェース
};
```

または，

```
template<class T,class Container>
class adapter_by_aggregation{
 Container x;
 public:
   // 定義し直したインターフェース
};
```

のような書き方をします．

継承を用いる前者のような方法は既存のコンテナに小さな機能を追加する場合に有効です．既存のコンテナの基本機能を利用しながら，異なるインターフェースを実装してアプリケーションに合ったものにしたいならば既存のコンテナをメンバとして持つ後者の方法が有効です．

インターフェースの再定義ができるため，一般には後者のほうが使われることが多いようです．

読者の皆さんも新しい種類のコンテナが欲しいと思っても，自分で作らずに既存コンテナをアダプタで拡張すると良いと思います．

● stack，queue

STLにはアダプタで実装されたstack，queue，priority_queueという三つのコンテナが含まれています．ここではstackとqueueの使用例をリスト10に，出力結果を図13に示します．

アルゴリズム

STLでは簡単なものから複雑なものまでさまざまなアルゴリズムがライブラリとして提供されています．ここではその中からcopy，sort，binary_searchに絞ってサンプルプログラムを使って紹介します（STLのアルゴリズムの説明をする場合，一番最初にfor_eachについて解説されることが多いが，筆者はあえてその方法はとらない）．

● アルゴリズム利用の基本

リスト11にcopyとsortおよびbinary_searchアルゴリズムの使用例を，出力結果を図14に示します．アルゴリズムライブラリを利用するにはリスト11の6行目のように<algorithm>というヘッダファイルを読み込みます．STLでは，アルゴリズム関数の基本型は以下のようになります．

```
template<class Iter>algo(Iter first,
                          Iter last,その他の引き数..);
```

<algorithm>の中の関数定義には上記のIter部分の型名が，

Input Iterator
Output Iterator
Forward Iterator
Bidirectional Iterator
Random Access Iterator

いずれかになっており，イテレータカテゴリがはっきりわかるようになっています．利用する際には<algorithm>の中身を見て確認すると良いでしょう．

● プログラムの流れ

リスト11，79行～129行目のmain関数の流れを説明します．ここではvector，list，dequeという3種類の列コンテナに対して，それぞれ疑似乱数をいくつか格納し，大小順に並べ替え，そしてbinary_search系のアルゴリズムを使って要素の検索を行っています．各コンテナに対する操作法はほとんど同じなので，とくに違いがない場合は98行～109行目のvectorの例を用いて説明します．

● 疑似乱数のコンテナへのコピー

リスト11の88行目では，21行～22行目で簡易的に作った疑似乱数配列tableの内容をcopyアルゴリズムを使ってコンテナにコピーしています．copyアルゴリズムは，

```
template<class InputIterafor,
                  class OutputIterator>
```

〔リスト10〕stackとqueueの使用例

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

////////////////////////////////////////////////////////
void Stack1() {
  int i;

  // 宣言 ////////////////////////////////////////////////
  stack<double> x;

  // 要素の追加 //////////////////////////////////////////
  for(i=0;i<10;i++) {
    x.push(i);
  }

  // 要素の参照と削除 ////////////////////////////////////
  while(!x.empty()) {
    cout << x.top() << " ";
    x.pop();
  }
  cout << endl;
}

////////////////////////////////////////////////////////
void Queue1() {
  int i;

  // 宣言 ////////////////////////////////////////////////
  queue<double> x;

  // 要素の追加 //////////////////////////////////////////
  for(i=0;i<10;i++) {
    x.push(i);
  }

  // 要素の参照と削除 ////////////////////////////////////
  while(!x.empty()) {
    cout << x.front() << " ";
    x.pop();
  }
  cout << endl;
}

////////////////////////////////////////////////////////
int main() {
  Stack1();
  Queue1();
  return 0;
}

////////////////////////////////////////////////////////
```

〔図13〕リスト10の出力結果

```
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
```

〔リスト 11〕sortと binary_searchアルゴリズムの使用例

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

/////////////////////////////////////////////////////////
template<class InputIterator>
void Display1(const string& comment
            ,InputIterator first,InputIterator last) {
  cout << comment << " ";
  while(first!=last) {
    cout << *first++ << " ";
  }
  cout << endl;
}

/////////////////////////////////////////////////////////
static double table[]
  = { .6, .9, .8, .5, .9, .2, .4, .1, .8, .3, .9, .3, 0};

double Randgen1() {
  static int index=0;
  double result = table[index++];
  if(0.0==table[index]) index = 0;
  return(result);
}

/////////////////////////////////////////////////////////
template<class Container>
void Assign1(Container& x,int n) {
  for(int i=0;i<n;i++) {
    x.push_back(Randgen1());
  }
}

/////////////////////////////////////////////////////////
template<class OutputIterator>
void FillRandom1(OutputIterator x,int n) {
  for(int i=0;i<n;i++) {
    *x++ = Randgen1();
  }
}

/////////////////////////////////////////////////////////
template<class RandomAccessIterator>
void Binary_search1(RandomAccessIterator first
                   ,RandomAccessIterator last,double val) {

  if(binary_search(first,last,val))
    cout << val << " exists" << endl;
  else
    cout << val << " does not exist" << endl;

  RandomAccessIterator position =
                              lower_bound(first,last,val);
  if(position!=last) {
    cout << "The first element equal or greater than "
         << val << " is " << *position << endl;
  }

  position = upper_bound(first,last,val);
  if(position!=last) {
    cout << "The first element greater than " << val
         << " is " << *position << endl;
  }

  pair<RandomAccessIterator,RandomAccessIterator>
    range = equal_range(first,last,val);
  cout << "List of elements equal to " << val << " are ";
  while(range.first != range.second) {
    cout << *range.first++ << " ";
  }
  cout << endl;
}

/////////////////////////////////////////////////////////
int main() {
  int i;

  /////////////////////////////////////////////////////
  cout << "### vector ###" << endl;
  vector<double> vd(12);

  //Assign1(vd,12);
  //FillRandom1(vd.begin(),12);
  copy(&table[0],&table[12],vd.begin());
  Display1("代入後",vd.begin(),vd.end());

  sort(vd.begin(),vd.end());
  Display1("sort後",vd.begin(),vd.end());

  Binary_search1(vd.begin(),vd.end(),0.5);


  /////////////////////////////////////////////////////
  cout << "### list ###" << endl;
  list<double> ld;

   //Assign1(ld,12);
   //FillRandom1(front_inserter(ld),12);
   copy(&table[0],&table[12],front_inserter(ld));
   Display1("代入後",ld.begin(),ld.end());

   ld.sort();
   Display1("sort後",ld.begin(),ld.end());

   Binary_search1(ld.begin(),ld.end(),0.3);


   /////////////////////////////////////////////////////
   cout << "### deque ###" << endl;
   deque<double> dd;

   //Assign1(dd,12);
   //FillRandom1(back_inserter(dd),12);
   copy(&table[0],&table[12],back_inserter(dd));
   Display1("代入後",dd.begin(),dd.end());

   sort(dd.begin(),dd.end());
   Display1("sort後",dd.begin(),dd.end());

   Binary_search1(dd.begin(),dd.end(),0.7);


   /////////////////////////////////////////////////////
   return 0;
}

/////////////////////////////////////////////////////////
```

〔図14〕リスト 11の出力結果

```
###vector###
代入後 0.6 0.9 0.8 0.5 0.9 0.2 0.4 0.1 0.8 0.3 0.9 0.3
sort後 0.1 0.2 0.3 0.3 0.4 0.5 0.6 0.8 0.8 0.9 0.9 0.9
0.5 exists
The first element equal or greater than 0.5 is 0.5
The first element greater than 0.5 is 0.6
List of elements equal to 0.5 are 0.5
###list###
代入後 0.3 0.9 0.3 0.8 0.1 0.4 0.2 0.9 0.5 0.8 0.9 0.6
sort後 0.1 0.2 0.3 0.3 0.4 0.5 0.6 0.8 0.8 0.9 0.9 0.9
0.3 exists
The first element equal or greater than 0.3 is 0.3
The first element greater than 0.3 is 0.4
List of elements equal to 0.3 are 0.3 0.3
###deque###
代入後 0.6 0.9 0.8 0.5 0.9 0.2 0.4 0.1 0.8 0.3 0.9 0.3
sort後 0.1 0.2 0.3 0.3 0.4 0.5 0.6 0.8 0.8 0.9 0.9 0.9
0.7 does not exist
The first element equal or greater than 0.7 is 0.8
The first element greater than 0.7 is 0.8
List of elements equal to 0.7 are
```

```
OutputIterator
copy(InputIterator first,
     InputIterator last,
     OutputIterator result);
```

と定義されており，第1，第2引き数にコピー元のコンテナの最初の要素と最後+1の要素を示すイテレータを，第3引き数にコピー先を示すイテレータを与えます．

コピー元は，21行～22行目で宣言されたdouble型の配列で，これの先頭と末尾+1のアドレスを&table[0]，&table[12]として与えています．

STLアルゴリズムでは，このように配列もコンテナの一種として，そのアドレスを示すポインタをそのままイテレータとして受け取ることができます．コピー先は84行目で宣言されているvectorコンテナvdで，あらかじめ疑似乱数を格納する領域として12個の要素を持っています．88行目では第3引き数にvd.begin()を与えているので，すでに準備した領域に先頭から順番に要素をコピーしていきます．

● ここに技あり！　挿入イテレータ

リスト11の88行目ではすでに準備された領域にコピーしましたが，STLアルゴリズムのすごいところは，同じcopyアルゴリズムを使って新たな要素を追加しながらコピーを行うこともできる点です．

リスト11の103行目ではcopyの第3引き数にfront_inserter(ld)を与えています．コピー先のldは99行目で要素が一つもないコンテナとして宣言されています．front_inserterという関数は引き数として与えられたコンテナに対してpush_frontを使って要素を追加するfront_insert_iteratorを生成します．そして，それを受け取ったcopyアルゴリズムは，コンテナの先頭位置に要素を追加していきます．

同様に118行目ではcopyの第3引き数はback_inserter(dd)です．back_inserterという関数は引き数として与えられたコンテナに対してpush_backを使って要素を追加するback_insert_iteratorを生成します．118行目のcopyは，114行目で要素が一つもないコンテナとして宣言されたddの末尾に要素を追加して行きます．

一見単純に見えるcopyアルゴリズムですが，高度に抽象化されたイテレータを用いることによって使える場面が思いのほか広いのです．従来のプログラミングテクニックではリスト11の32行～37行目に示したAssign1関数のように，コンテナそのものを引き数として受け取り，どの位置にどのようにコピー，または挿入するかということを固定したライブラリしか作れませんでした．

STLの考え方では40行～45行目のFillRandom1関数のようにイテレータを引き数として受け取って操作します．

与えられるイテレータの種類によって，

a) すでにある領域への上書き
b) 先頭への追加
c) 末尾への追加
d) 任意位置への挿入

といった動作を切り替えることができます．

● sortを使った並べ替え

リスト11の91行目と121行目ではコンテナに格納された内容を大小順に並べ替えるsortアルゴリズムを使っています．sortアルゴリズムはクイックソートという高速なアルゴリズムを使っていて，ランダムアクセスができるコンテナにしか適用できません．

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first,
                 RandomAccessIterator last);
```

vectorとdequeはランダムアクセス可能なのでsortが使えますが，双方向走査しかできないlistにはそれが使えないので，特別にメンバ関数としてsortが準備されています．リスト11の106行目のように，アルゴリズムとコンテナの組み合わせの中には利用できないものもあるので注意が必要です．

● binary_search

実は二分検索を利用したアルゴリズムは，

binary_search，lower_bound，upper_bound，equal_range

の4種類があります．いずれもコンテナ内の要素がすでに大小順に並べ替えられていることを前提とします．

リスト11の52行目で使用しているbinary_searchは，検索する値に一致するものの有無をbool値で返します．57行目のlower_boundは，検索する値に一致する最初の要素の位置を示すイテレータを返します．63行目のupper_boundは，検索する値に一致する最後の要素+1の位置を示すイテレータを返します．70行目のequal_rangeはlower_boundとupper_boundの結果を合わせてイテレータのペアとして返します．

一般にbinary_searchもランダムアクセスが必要になるので，一見，listには適用できないように思われますが，実はbinary_searchは，

```
template<class ForwardIterator,class T>
  bool biary_search(ForwardIterafor first,
       ForwardIterator last ,const T&value);
```

のように定義されておりlistでも利用できます．

これはイテレータの移動よりも要素の比較のほうが時間がかかる場合があることを想定したためです．binary_search系のアルゴリズム中ではイテレータの種類を判定し，Random Access IteratorならばそのままランダムアクセスをADD行い，Forward IteratorまたはBidirectional Iteratorならば内部的には順次移動を行うようになっています．

Part 3　STL 利用に関する考察

なぜアルゴリズムの利用が進まないか

STL はもともと汎用アルゴリズムの開発を目指した技術です．それにもかかわらず，現実には単なるコンテナライブラリと思われていることが多いようです．汎用的なコンテナライブラリを提供したことはそれだけでも大きな功績ですが，では，なぜ本来の目的だった汎用アルゴリズムの利用が思ったほど進まないのでしょうか．決まった答があるわけではありませんが，ここで筆者なりの意見を述べます．

利用が進んでいる部分もそうでない部分も含めて，STL はソフトウェアの汎用部品化という目標に対して大きな成功を収めています．問題は，現在のソフトウェア開発技法に，部品化によって大きく改善できるところとそうではないところがある点だと思います．部品化も過ぎたるは及ばざるが如しで，細切れにしすぎてかえって使いにくくなる場合もあります．

● for_each と関数オブジェクトの問題点

例として，リスト 12とリスト 13に，簡単なプログラムを for_each と関数オブジェクトを使う方法と使わない方法とで書いて比較してみました．リスト 12の 12行〜 18行目の simple1という関数では vector コンテナに値をいくつか入れた後，その結果を 16行目で for_each アルゴリズムと print_value 関数オブジェクトを使って表示しています．これと同じことをリスト 13の 7行〜 15行目の，同じく simple1 という関数で行っています．ここでは 11行〜 13行目で for 文を使って全要素の表示を行っています．

小さなプログラムなので二つのやり方に大きな優劣の違いはないように思えます．しかし，大きなプログラムで for_each を多用した場合，筆者の経験ではプログラムが細分化されすぎて非常にわかりにくいものになってしまいました．もともと，2〜 3行で表現できることをわざわざ部品に分離しても読みにくくなるだけで利点はあまりないのではないでしょうか．

また，変更の容易性にも疑問があります．たとえば，前記の simple1 の表示ループを少し変更して，ついでに要素の総和を求めるように変更した modified1 関数を見てみましょう．

リスト 12の場合，総和を求めるという動作を同じ print_value 関数オブジェクトにやらせられないことはないのですが，面倒な書き方をしなくてはなりません．素直に総和を求めると，26行目のように accumulate アルゴリズムを使うことになります．この場合，コンテナの走査を for_each と

〔リスト 12〕for_each と関数オブジェクトを使って記述した例

```
 1: #include <iostream>
 2: #include <vector>
 3: #include <algorithm>
 4: #include <numeric>
 5: using namespace std;
 6:
 7: ////////////////////////////////////////////////////////
 8: void print_value(double x) {
 9:   cout << x << "  ";
10: }
11:
12: void simple1() {
13:   vector<int> a;
14:   for(int i=0;i<10;i++) a.push_back(i);
15:
16:   for_each(a.begin(),a.end(),print_value);
17:   cout << endl;
18: }
19:
20: ////////////////////////////////////////////////////////
21: void modified1() {
22:   vector<int> a;
23:   for(int i=0;i<10;i++) a.push_back(i);
24:
25:   for_each(a.begin(),a.end(),print_value);
26:   int sum = accumulate(a.begin(),a.end(),0);
27:   cout << "sum=" << sum << endl;
28: }
29:
30: ////////////////////////////////////////////////////////
31: int main() {
32:   simple1();
33:   modified1();
34:   return 0;
35: }
36:
37: ////////////////////////////////////////////////////////
```

〔リスト 13〕for_each と関数オブジェクトを使わないで記述した例

```
 1: #include <iostream>
 2: #include <vector>
 3: #include <algorithm>
 4: using namespace std;
 5:
 6: ////////////////////////////////////////////////////////
 7: void simple1() {
 8:   vector<int> a;
 9:   for(int i=0;i<10;i++) a.push_back(i);
10:
11:   for(vector<int>::iterator i=a.begin();i!=a.end();++i) {
12:     cout << *i << "  ";
13:   }
14:   cout << endl;
15: }
16:
17: ////////////////////////////////////////////////////////
18: void modified1() {
19:   vector<int> a;
20:   int sum=0;
21:   for(int i=0;i<10;i++) a.push_back(i);
22:
23:   for(vector<int>::iterator i=a.begin();i!=a.end();++i) {
24:     cout << *i << "  ";
25:     sum += *i;
26:   }
27:   cout << "sum=" << sum << endl;
28: }
29:
30: ////////////////////////////////////////////////////////
31: int main() {
32:   simple1();
33:   modified1();
34:   return 0;
35: }
36:
37: ////////////////////////////////////////////////////////
```


Information ── オラクル，T-Engine 対応の小型データベース「Oracle Life for T-Engine」を開発
日本オラクル(株)は，T-Engine 対応の組み込み向け小型データベースソフトウェア「Oracle Life for T-Engine」を開発し，製品化を目指している．本製品は，データベース機能との連携も行える．

accumulateで2回行うことになり，処理が無駄になります．

一方，リスト13のmodified1では単に sum += *i'という命令をforループ内に追加するだけです．

● 複雑なアルゴリズムを中心に利用

このように，テキストエディタでプログラムを作成するという現在のソフトウェア開発形態では，for_eachのような粒の小さなアルゴリズムはソースコードの過度な分断をもたらしてプログラムを非効率的にしてしまうことがあります．

リスト12のような方法は，ソフトウェア開発技法が変わって，たとえばGUIで部品をつなぎ合わせるようなものにならなければ力を発揮しないのではないかと思います．STLアルゴリズムの説明はたいがいはfor_eachから始まりますが，そこで利点を実感できないために勉強を止めてしまう人がかなりいるのではないかと思います．

筆者は，単純で粒が小さなアルゴリズムはとりあえず無視して，複雑で独自開発が難しいものを中心に利用を始めるのが良いと思います．部品化というコンセプトは若干後退しますが，現実的なやりかたではないかと思います．

コンテナの選択

STLの解説書を読むとコンテナの種類や構造によって，実行効率の良い操作，悪い操作があると書いてあります．そして，どのような操作を行うかを考えて効率の良いコンテナを選ぶべしと書いてあります．

たしかに，それはまっとうな指示ではあるのですが，筆者の考えでは実際にコンテナを利用する場面を考えると，解説書におけるコンテナ選択と効率の考察は限定的で，あまり実用的でないと思います．非常に簡単なプログラムを作る場合にはほとんど気にする必要はありませんし，逆に本格的なプログラムを開発する場合にはもっと深い考察が必要です．

要は以下の二つの不等式が成り立つことが重要なのです．

a）使用頻度または回数×一回当たりの実行速度
　　＜許容される処理時間

b）要素数×要素当たりのメモリ使用量
　　＜許容されるメモリリソース量

● 簡単なプログラムの場合

たとえ効率の悪い処理を行ったとしても，プログラムが簡単で，上記のa），b）が確実に成り立つような場合はコンテナの種類や実行効率を気にする必要はありません．むしろ簡単に開発できることを優先すべきです．

● 本格的なプログラムの場合

本格的なプログラムを開発するとき，いちばん注意が必要なのは要素の数です．STLコンテナはプロセス上のメモリ空間に内容をすべて展開することを前提としています．そのため，b）の条件が崩れることが予想されるならば，STLコンテナを単純に使うことはできなくなります．データのハードディスクへの

〔リスト14〕コンパイルエラーの可能性のある例

```
 1: // 悪い例，vectorのイテレータは必ずしもポインタとは限らない
 2: #include <iostream>
 3: #include <vector>
 4: using namespace std;
 5:
 6: int main() {
 7:   vector<double> x;
 8:   int i;
 9:   for(i=0;i<10;i++) x.push_back(i*1.1);
10:
11:   double *ary = x.begin(); // コンパイルできない場合がある
12:   for(i=0;i<10;i++) cout << ary[i] << " ";
13:   cout << endl;
14:
15:   return 0;
16: }
```

退避，市販データベースの利用などを考える必要があります．

もちろん，その中ではSTLコンテナを利用する場面があると思いますが，むしろ，問題をa），b）の不等式を満たすような大きさに分解する設計力が重要です．

このあたりはSTLの解説書に書いてあることにとらわれず，各自で判断するしかありません．

パフォーマンスに余裕がある場合には開発の簡便性で選択し，そうでないときにはもっと広い範囲の検討を行い，単なるコンテナの選択よりは全体の構想設計を重視します．

STL利用の注意

最後にSTLを利用する際のちょっとした注意点を述べます．

● 移植性

STLを使ったプログラムを異なるコンパイラに移植する場合，必ずANSI/ISO C++標準かどうかを調べ，きちんとしたSTLの解説書を読んで，公開されている使い方をすることが重要です．STLはC++のあらゆる細部を利用しているため，コンパイラのほうも余裕がありません．正式に定義されていない使い方をすると，あるコンパイラではたまたま動作するものの，他のコンパイラではコンパイルすらできないことがあります．

ここでは，その一例としてvectorコンテナのイテレータを取り上げます．多くのコンパイラではvectorのイテレータは要素型へのポインタとして実現されています．

リスト14の11行目ではそのことを利用してbegin()というイテレータを返すメンバ関数の結果を要素型のポインタである'double *ary'に代入しています．現在出回っている80%以上のコンパイラはこのような記述を受け付けてしまいます．

ところが，ANSI/ISO C++ではvectorのイテレータは実装に依存する任意の型で良いと定められているため，リスト14の11行目はコンパイルエラーとなる場合があります．

● 追加，削除後はイテレータを再取得

コンテナの中身を参照しながら，同時に追加や削除を行いたい場合には注意が必要です．というのは，イテレータの内容はコンテナに追加や削除を行う前後で保存されることが保障され

〔リスト 15〕イテレータの内容の保存は保障されない

```
 1: #include <iostream>
 2: #include <vector>
 3: using namespace std;
 4:
 5: ///////////////////////////////////////////////////////////
 6: void bad1() {
 7:   int i;
 8:   vector<int> x;
 9:   for(i=0;i<5;i++) x.push_back(i);
10:
11:   // ある状態で begin(), end() を取得
12:   vector<int>::iterator first = x.begin();
13:   vector<int>::iterator last  = x.end();
14:   vector<int>::iterator p;
15:   for(p=first;p!=last;++p) cout << *p << " ";
16:   cout << endl;
17:
18:   // 要素を追加
19:   for(i=5;i<10;i++) x.push_back(i);
20:
21:   // !!! 追加前に取得したイテレータをそのまま使用 !!!
22:   for(p=first;p!=last;++p) cout << *p << " ";
23:   cout << endl;
24: }
25:
26: ///////////////////////////////////////////////////////////
27: void good1() {
28:   int i;
29:   vector<int> x;
30:   for(i=0;i<5;i++) x.push_back(i);
31:
32:   // ある状態で begin(), end() を取得
33:   vector<int>::iterator first = x.begin();
34:   vector<int>::iterator last  = x.end();
35:   vector<int>::iterator p;
36:   for(p=first;p!=last;++p) cout << *p << " ";
37:   cout << endl;
38:
39:   // 要素を追加
40:   for(i=5;i<10;i++) x.push_back(i);
41:
42:   // 追加後に begin(), end() を再取得
43:   first = x.begin();
44:   last  = x.end();
45:   for(p=first;p!=last;++p) cout << *p << " ";
46:   cout << endl;
47: }
48:
49: ///////////////////////////////////////////////////////////
50: int main() {
51:   good1();
52:   bad1();
53:   return 0;
54: }
55:
56: ///////////////////////////////////////////////////////////
```

ていないからです．

たとえばリスト 15の6行～24行目のbad1という関数の中で，19行目の要素追加の前に取得したイテレータ first，lastをそのまま追加後の22行目で利用しています．このような使い方をしてもコンテナの種類や状況によっては所望の動作をしてしまうことがあります．要素追加の頻度が低い場合には，

〔リスト 16〕古いコンパイラで問題が起こる例

```
 1: #include <map>
 2: #include <vector>
 3: #include <list>
 4: #include <string>
 5: using namespace std;
 6:
 7: map<string,vector<int> > a1;
 8: map<string,vector<vector<int> > > a2;
 9: map<string,vector<vector<vector<int> > > > a3;
10: map<string,vector<vector<vector<list<int> > > > > a4;
11: vector<vector<vector<int> > > v1;
12:
13: template<class T>
14: void f(T& x) {
15:   T::iterator i;
16:   for(i=x.begin();i!=x.end();++i) {
17:     //...
18:   }
19: }
20:
21: int main() {
22:   f(v1);
23:   return 0;
24: }
```

誤動作がごくまれにしか発生しないため，たいへん見つけにくい不具合を埋め込むことになってしまいます．STLを使う場合には，たまたま動作したからといって油断せず，きちんと正式な動作仕様と制限を確認するようにしましょう．

● コンパイラの制限

最近はコンパイラ技術が向上して，かなり複雑なSTLの使い方をしてもコンパイルできるようになりました．しかし，古いコンパイラの中にはリスト 16の7行～11行目に示したようなコンテナの入れ子や15行目に示したようなごく普通に見える宣言文をコンパイルできないものもあります．STLは一見単純に見えても，中身は複雑怪奇，コンパイラメーカー泣かせの代物です．一気に複雑な記述をせず，コンパイラのご機嫌をうかがいながら少しずつやりたいことを実現していくと良いでしょう．

おわりに

STLはたいへん有用で汎用的なライブラリなので，C++でプログラムを書く人にはかならずマスターしてほしいと思います．しかし，すべて満遍なく勉強すると膨大な時間がかかるので，本章で説明したようにlist，deque，map，それにsort，binary_searchなど，いくつかの重要なコンテナやアルゴリズムから始めると良いでしょう．

ごとう・まさはる　アジレント・テクノロジー（株）

Chapter 4

使い慣れた言語で，今日から役立つ

C言語で使える コンテナライブラリ

曽田 哲之

ここまでの章では，C++言語でのテンプレートプログラミングを紹介してきた．しかし，組み込み分野では，C言語を用いて開発することも多いと思われる．そこで本章では，STLの諸機能の中でももっとも有用であろうと思われるコンテナライブラリとしての側面について，同様の機能を提供するC言語向けライブラリを紹介する．（筆者）

復習：STLのコンテナライブラリ

STLは，次のような特徴をもったコンテナライブラリでした．

●型に関する安全性

テンプレートを用いて実現しているため，コンテナライブラリとして利用した場合も，型検査が行われます．

●高速

同じくテンプレートを用いて実現しているため，関数ポインタや仮想関数呼び出しを経由する必要がありません．このためオーバヘッドが少なく，また，コンパイラによるインライン展開などの恩恵を受けることができるため，高速です．

●イテレータを介した，汎用アルゴリズムとの組み合わせ

これらのうち，汎用アルゴリズムとの組み合わせ機能については，残念ながら今回紹介するC言語のライブラリでは満足できません．これは，アルゴリズムとコンテナを仲介するイテレータが，本章で紹介するライブラリには存在しないことがおもな理由です．

それでも，マクロ展開と組み合わせれば同様な機能の実現は不可能ではありませんが，マクロを使った場合，開発やデバッグの手間なども増えてしまうため，今回は考えないことにします．この機能を使いたい場合にはC++とSTLを導入すべきでしょう．

本章で紹介するコンテナライブラリは，GNOME/GTKプロジェクトが中心になって開発しているglibと，BSD系フリーUNIXプロジェクトが開発しているqueue.hおよびtree.hです．

glibの機能と解説

glibは，コンテナライブラリ，メモリ管理ライブラリとしての範ちゅうにとどまらず，さまざまなアプリケーションから使える便利な関数を多く含んだライブラリで，次のサイトから入手できます．

ftp://ftp.gtk.org/pub/gtk/v2.2/glib-2.2.3.tar.gz

glibのライセンスはLGPLなので，製品のエンドユーザーに対して，glibソースコードを入手できることを保証する必要がありますが，glibを組み込んだバイナリを配布する場合に，そのソースコードのすべて（glib以外の部分）を公開する必要はありません．詳しいライセンスに関してはLGPLのライセンス条文を参照してください．リファレンスマニュアルは，http://developer.gnome.org/doc/API/2.0/glib/index.htmlか，またはソース配布中のdocs/reference/glib/html/index.htmlから参照できます．

コンテナライブラリとしてのglibは，単方向リスト（g_slist），双方向リスト（g_list），末尾へのポインタ付き双方向リスト（g_queue），サイズ拡張可能な配列（g_array，g_ptr_array，g_byte_array），ハッシュテーブル（g_hash_table），平衡探索木（g_tree）などなど，数多くのデータ型を提供しています．これらは，gpointerという型を経由してアクセスするため，型に関する安全性はありません．また，リストをたどる以外のほとんどの機能は関数呼び出しを用いて利用するので，STLに比べると速度的にも若干劣る場合があると思われます．

では，使用例を見てみましょう．

●g_slist

g_slistは，単方向リストを実現しています．

リスト1～リスト3は，標準入力から読んだ各行を，g_slist_prepend()で変数listが指すリストの先頭につけ加えます．subr.hはリストに含まれていませんが，本章のプログラム中で定義されている共通関数のプロトタイプ宣言を集めたヘッダです．プログラムの最後では，g_slist_foreach()を使って，リストを構成する各ノードについて，先頭から順番にprint_list_node()を呼び出して表示します．リストの末尾ではなく先頭につけ加えていくため，結果として，このプログラムは，入力ファイルの各行を逆順に表示することになります．

型GSListは，次に示すように定義されており，リスト構造の先頭を指すポインタ型と，リストの各ノードを表す型の両方

〔リスト 1〕subr_strdup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

char *
estrdup(const char *src)
{
    char *dst = strdup(src);

    if (dst == NULL) {
        fprintf(stderr, "no memory¥n");
        exit(EXIT_FAILURE);
    }
    return dst;
}
```

〔リスト 2〕subr_line.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

char *
get_line_alloc(FILE *fp)
{
    char buf[BUFSIZ];
    int len;

    if (fgets(buf, sizeof buf, fp) == NULL)
        return NULL;
    len = strlen(buf);
    if (len > 0 && buf[len - 1] == '¥n')
        buf[len - 1] = '¥0';
    return estrdup(buf);
}
```

〔リスト 3〕glib_slist.c

```
#include <stdio.h>
#include "subr.h"
#include <glib.h>

void
print_list_node(gpointer data, gpointer user_data)
{
    char *s = data;

    printf("%s¥n", s);
}

int
main()
{
    GSList *list = NULL;
    char *line;

    while ((line = get_line_alloc(stdin)) != NULL) {
        list = g_slist_prepend(list, line);
    }
    g_slist_foreach(list, print_list_node, NULL);
    return 0;
}
```

の目的で使われます．

```
typedef struct _GSList GSList;
struct _GSList {
    gpointer data;
    GSList *next;
};
```

プログラム中の，

```
g_slist_foreach(list, print_list_node, NULL);
```

という行は，

```
for (e = list; e != NULL; e =
                        g_slist_next(e)) {
    printf("%s", (char *)e->data);
}
```

のように書き下すこともできます．

このプログラムで，g_slist_prepend()の代わりにg_slist_append()を用いると，入力行はリストの末尾に追加され，最終的な表示は正順になります．

しかし，g_slist_append()の利用は，じつはあまり好ましくありません．というのはg_slistがリスト末尾へのポインタを保持していないため，g_slist_append()は，リスト中のすべてのノードをたどらないと末尾にノードを追加することができないからです．このため，リスト中のノード数をNとすると，g_slist_append()の一回の呼び出しには$O(N)$の実行時間がかかってしまいます．したがって，g_slist_append()を利用するようにしたバージョンをN行のファイルに対して実行すると，プログラム全体の実行には$O(N^2)$の時間コストがかかります〔g_slist_prepend()の場合，1回のコストは$O(1)$なので，プログラム全体の実行コストは$O(N)$で済む〕．

そのため，リスト末尾へのアクセスが必要な場合にはg_slistは用いず，glibならg_queueを使うか，あるいはBSDのqueue.hにあるSTAILQを用いるのが望ましい方法です．

このように，ある操作を行うインターフェースが用意されているからといって，必ずしもその操作の効率が良いとは限らないので注意が必要です．とくにg_slistの場合，g_slist_insert_before()，g_slist_remove_link()といった操作まで，$O(N)$のコストがかかります．

● g_list

リスト1のGSListをGListに書き換え，g_slistをg_listに書き換えると，双方向リストを用いたコンテナになります．リスト構造の先頭を指すポインタ型およびリストの各ノードを表す型として用いられるGListは，次のような構造体です．

```
typedef struct _GList GList;
struct _GList {
    gpointer data;
    GList    *next;
    GList    *prev;
};
```

単方向リストとの違いは，g_list_previous()を用いてリストを逆順にたどることが可能なことと，g_list_insert_before()，g_list_remove_link()が定数コスト：$O(1)$になる点です．g_list_append()のコストが$O(N)$である点は単方向リストの場合と変わりません．

● g_queue

g_queueは，末尾へのポインタ付きの双方向リストです．STLのlistコンテナと機能的に対応します．g_queueは，先頭要素および末尾要素へのポインタを下記のGQueue型として保持しています．

```
typedef struct _GQueue GQueue;
struct _GQueue
{
    GList *head;
    GList *tail;
    guint length;
};
```

リストの各ノードは，単なる双方向リストと同様に，GList型を用いています．リスト4は，リスト1をg_queueを用いて書き直したものです．g_slistおよびg_listの_prepend()，_append()に対応する関数は，g_queueではそれぞれg_queue_push_head()，g_queue_push_tail()という名称になっています．また，先頭および末尾からノードを取り除くg_queue_pop_head()，g_queue_pop_tail()関数も用意されており，このプログラムでは，それらを表示の際に使っています．このためリスト1と異なり，表示時にリストを破壊し，ノードを順次捨てていくことになります．

g_queueにはなぜか_foreach()関数が用意されていません．しかし，各ノードはGList型なので，表示部を次のようにすれば，リスト1と同様に，リストの内容を変更せずに表示することができます．

```
g_list_foreach(list->head, print_list_node,
                                      NULL);
```

当然，これは次のように書くこともできます．

```
for (node = list->head; node != NULL;
                  node = g_list_next(node)) {
    printf("%s¥n", (char *)node->data);
}
```

しかし，このようにg_list用の関数をg_queueに対して用いることが，glibの開発者から見て推奨できる用法であるのかどうかはよくわかりません．

g_queueにはリスト末尾へのポインタがあるので，リストへのデータ挿入処理においてg_queue_push_head()の代わりにg_queue_push_tail()を呼ぶようにするか，あるいはリストの表示処理において，g_queue_pop_head()の代わりにg_queue_pop_tail()を呼ぶようにすれば，表示を正順で行うことができます．g_queue用に用意されている操作はすべて合理的なコストで実行可能です(g_queue_free()以外はすべて定数コスト)．

● サイズ拡張可能な配列
(g_array，g_ptr_array，g_byte_array)

STLのvectorに相当する機能です．これについては，glibリファレンスマニュアルのトップページ中の"List of Examples"にあるGAarray，GPtrArray，GByteArrayの項目に利用例があります．

〔リスト4〕glib_queue.c

```
#include <stdio.h>
#include <stdlib.h>
#include "subr.h"

#include <glib.h>

int
main()
{
    GQueue *list = g_queue_new();
    char *line;

    while ((line = get_line_alloc(stdin)) != NULL) {
        g_queue_push_head(list, line);
    }
    while  ((line = g_queue_pop_head(list)) != NULL) {
        printf("%s¥n", line);
        free(line);
    }
    return 0;
}
```

● g_hash_table

ハッシュ表です．C++言語標準を策定する際には時間的制約から，STLにはハッシュ表の定義が含まれませんでした注1．しかし現実の応用では，ハッシュ表はもっとも良く利用されるコンテナ型の一つでしょう．

ハッシュ表は，STLの標準コンテナであるset，multiset，map，multimapのような，連想記憶型の検索を提供します．ただし，STLの標準連想コンテナとは異なり，データをsortされた順序でアクセスする機能はありません．

その代わり，STLの標準連想コンテナでは連想アクセスに(データ件数をNとして)，$O(\log N)$の時間コストがかかるのに対し，ハッシュ表を使うと$O(1)$とより高速に検索できます．ただし，STLの標準連想コンテナの場合，最悪の場合でも$O(\log N)$しか時間がかからないことが保証されていますが，ハッシュ表は平均的な性能として$O(1)$でふるまうだけであり，最悪の場合には$O(N)$と，リスト構造を使うのと変わらない性能に落ちてしまいます．

とはいっても適切なハッシュ関数を選択していれば，現実にはほぼ常に$O(1)$に近い性能を示します．ただし，意図的に最悪の場合を狙われた場合にはこの限りではないので，そういう場合も考慮せざるを得ない状況では，ハッシュ表ではなく平衡探索木を用いたほうが良いでしょう．AWKやPerlのようなスクリプト言語が提供する連想配列もハッシュ表を用いて実現されています．

リスト5〜リスト7は，ハッシュ表の使用例です．この例では，コマンド行で指定したファイルを読み込み，そのファイルに含まれる単語の出現頻度を記録します．その後，標準入力待ちの状態に移り，標準入力から「list」と入力すると全単語と

注1：ただし，ほとんどのSTL実装は，非標準機能としてハッシュ表を含んでいる．

頻度の対応表，「lookup 単語」と入力すると指定した単語とその頻度を表示します．

glibでハッシュ表を表現する型はGHashTableです．リスト7のmap_init()にあるように，ハッシュ表の初期化時には，連想探索キーの比較関数と，探索キーからハッシュ値を求める関数の二つの関数を指定する必要があります．ここで指定しているword_hash()関数はPerl言語の連想配列の実装でも用いられているもので，ごく簡単なわりには良い性能を示すことが知られています．

なお，glibは標準的な文字列用比較関数およびハッシュ関数としてg_str_equalおよびg_str_hashをあらかじめ提供しているので，文字列に対するハッシュ表を作成する場合，実際にはこの例のように自分で比較関数とハッシュ関数を作成する必要はありません．ここで実装を示したのは単に例としてのものです．

glibは，文字列用以外にも，ポインタ値(gpointer)をそのまま連想キーとして使う場合と，整数値(gint)を連想キーとして使う場合について，比較関数とハッシュ関数を用意しています．これはそれぞれg_direct_equal()/g_direct_hash()と，g_int_equal()/g_int_hash()です．

ハッシュ表を用いた連想検索は，リスト7のword_add()やword_lookup()からわかるように，g_hash_table_lookup()関数で行います．また，ハッシュ表に新たにデータを追加するには，word_add()にあるようにg_hash_table_insert()を用います．

なお，リスト7では構造体word_frequencyに，連想キーをwordメンバ変数として含んでいます．実際の応用では，この例のように各ノードに連想キーも記憶しておくと便利なことが多いのですが，この例では実際にはwordメンバ変数をアクセスしていないので，省略してもかまいません．

● g_tree

平衡二分探索木です．STLの標準連想コンテナであるset，multiset，map，multimapに対応します．平衡化を行わない単なる二分木も，連想記憶を実現するための探索木として使えますが，その場合には二分木の左右の高さのバランスが崩れ，性能が極めて悪化することが多いので，汎用のライブラリの探索木として用いるには，平衡化が必須です(たとえばすでにsortされたデータを初期データとして二分探索木に与えると，すべてのノードが親ノードの右の子として連結されて，実質上リストと同様な構造に縮退してしまい，探索コストが$O(N)$になってしまう)．

ほとんどのSTLの実装では，平衡探索木として赤黒木を採用していますが，glibではAVL木を用いています．AVL木は，

〔リスト5〕subr_alloc.c

```
#include <stdio.h>
#include <stdlib.h>
#include "subr.h"

void *
emalloc(size_t size)
{
    void *p = malloc(size);

    if (p == NULL) {
        fprintf(stderr, "no memory¥n");
        exit(EXIT_FAILURE);
    }
    return p;
}
```

〔リスト6〕main_word.c

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

char *
get_word_alloc(FILE *fp)
{
    int c, len;
    char buf[BUFSIZ];

    while ((c = getc(fp)) != EOF && !isalnum(c))
        ;
    if (c == EOF)
        return (NULL);
    /* assert(isalnum(c)); */
    len = 0;
    do {
        if (len >= sizeof(buf) - 1) /* too long */
            break;
        buf[len++] = c;
    } while ((c = getc(fp)) != EOF && isalnum(c));
    buf[len] = '¥0';
    return estrdup(buf);
}

int
main(int argc, char **argv)
{
    FILE *word_fp;
    char *word, buf[BUFSIZ], cmd[BUFSIZ], arg[BUFSIZ];
    int cmdc;

    map_init();
    if (argc != 2) {
        fprintf(stderr, "missing input filename¥n");
        exit(EXIT_FAILURE);
    }
    if ((word_fp = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "cannot open %s¥n", argv[1]);
        exit(EXIT_FAILURE);
    }
    while ((word = get_word_alloc(word_fp)) != NULL) {
        word_add(word);
    }
    fclose(word_fp);

    while (fgets(buf, sizeof buf, stdin) != NULL) {
        cmdc = sscanf(buf, "%s %s", cmd, arg);
        if (cmdc == 0)
            continue;
        if (strcmp(cmd, "list") == 0) {
            word_list();
        } else if (strcmp(cmd, "lookup") == 0) {
            if (cmdc < 2) {
                fprintf(stderr, "missing word argument¥n");
            } else {
                word_lookup(arg);
            }
        } else {
            fprintf(stderr, "%s: unknown command¥n", cmd);
        }
    }
    return 0;
}
```

二分木のすべてのノードについて，左の子と右の子の高さの差がたかだか1しか違わないという基準で平衡化された二分木です．完全にバランスされた木に比べて，最悪でも45%しか木の高さが増えないことが知られています．

アルゴリズムの理論上では，平衡化のための木の回転操作の回数の違いで，赤黒木のほうが優れていると見なされることが多いのですが，AVL木のほうが木の高さ自体は低くなるため，とくに木へのデータの挿入や削除の回数よりも検索の回数のほうが多い場合には，AVL木の方が実際には高速であることもしばしばあります．

リスト8は，ハッシュ表の場合と同様に単語の出現頻度を数えるプログラムの例です．glibで平衡探索木を表現する型はGTreeです．リスト8のmap_init()にあるように，平衡探索木の初期化時には，連想探索キーの比較関数が必要です．ハッシュ表の場合の比較関数は，キーが等値かどうかを判別して真偽値を返す関数でしたが，平衡探索木の場合には，キーの大小関係を比較し，負の数・零・正の数の返り値で，それぞれ小・等価・大を表す関数が必要なので注意してください．

連想検索は，リスト8のword_add()やword_lookup()からわかるように，g_tree_lookup()関数で行います．データの追加は，word_add()からわかるようにg_tree_insert()を用います．

なお，リスト8でも構造体word_frequencyに，連想キーをwordメンバ変数として含んでいますが，この例では実際にはwordメンバ変数をアクセスしていないので，省略してかまいません．

● 整数の格納

ここでのプログラム例では，gpointerには記憶するデータへのポインタを格納してきました．しかし，記憶するデータの内容が整数一つの場合にはgpointerに直接整数を格納することもできます．この場合に使える整数型としては符合付整数型gint，符合なし整数型guint，C言語標準のsize_t相当のgsizeがあり，それぞれ，GPOINTER_TO_INT()，GINT_TO_POINTER()，GPOINTER_TO_UINT()，GUINT_TO_POINTER()，GPOINTER_TO_SIZE()，GSIZE_TO_POINTER()といったマクロで，gpointer型へ変換して使います．

BSD queue.h，tree.h

queue.hとtree.hは，NetBSD，FreeBSD，OpenBSDなどBSD系のOSに附属するヘッダファイルです．queue.hに関しては，Linuxにも古いバージョンが付属してきます．

このコンテナライブラリは，ヘッダ中のマクロのみで構成さ

〔リスト7〕glib_hash.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

#include <glib.h>

struct word_frequency {
    char *word;
    int count;
};

gboolean
word_equal(gconstpointer a, gconstpointer b)
{
    return strcmp((const char *)a, (const char *)b) == 0;
}

guint
word_hash(gconstpointer key)
{
    const char *s = key;
    unsigned int h = 0;

    while (*s != '¥0')
        h = h * 33 + *s++;
    return h + (h >> 5);
}

GHashTable *map;

void
map_init(void)
{
    map = g_hash_table_new(word_hash, word_equal);
}

void
word_add(char *word)
{
    struct word_frequency *node;
    gpointer found;

    found = g_hash_table_lookup(map, word);
    if (found != NULL) { /* already exists */
        node = found;
        node->count++;
        return;
    }

    node = emalloc(sizeof(*node));
    node->word = word;
    node->count = 1;
    g_hash_table_insert(map, word, node);
}

void
word_print(gpointer key, gpointer value, gpointer user_data)
{
    char *key_string = key;
    struct word_frequency *node = value;

    printf("%d¥t%s¥n", node->count, key_string);
}

void
word_list(void)
{
    g_hash_table_foreach(map, word_print, NULL);
}

void
word_lookup(char *word)
{
    struct word_frequency *node= g_hash_table_lookup(map, word);

    if (node == NULL) {
        fprintf(stderr,"%s: not found¥n", word);
    } else {
        word_print(word, node, NULL);
    }
}
```

〔リスト 8〕glib_tree.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

#include <glib.h>

struct word_frequency {
    char *word;
    int count;
};

gint
word_compare(gconstpointer a, gconstpointer b)
{
    return strcmp((const char *)a, (const char *)b);
}

GTree *map;

void
map_init(void)
{
    map = g_tree_new(word_compare);
}

void
word_add(char *word)
{
    struct word_frequency *node;
    gpointer found;

    found = g_tree_lookup(map, word);
    if (found != NULL) { /* already exists */
        node = found;
        node->count++;
        return;
    }
    node = emalloc(sizeof(*node));
    node->word = word;
    node->count = 1;
    g_tree_insert(map, word, node);
}

void
word_print(gpointer key, gpointer value, gpointer user_data)
{
    char *key_string = key;
    struct word_frequency *node = value;

    printf("%d¥t%s¥n", node->count, key_string);
}

gboolean
word_tree_traverse(gpointer key, gpointer value,
                   gpointer user_data)
{
    word_print(key, value, user_data);
    return (FALSE);
}

void
word_list(void)
{
    g_tree_foreach(map, word_tree_traverse, NULL);
}

void
word_lookup(char *word)
{
    struct word_frequency *node = g_tree_lookup(map, word);

    if (node == NULL) {
        fprintf(stderr,"%s: not found¥n", word);
    } else {
        word_print(word, node, NULL);
    }
}
```

れており，リンクするライブラリ関数などはありません．下記のftpサイトなどから入手可能です．

ftp://ftp.NetBSD.org/pub/NetBSD/NetBSD-current/src/sys/sys/

マニュアルは下記にqueue.3，tree.3というファイル名で置かれています．

ftp://ftp.NetBSD.org/pub/NetBSD/NetBSD-current/src/share/man/man3/

このマニュアルはgroffのmdoc形式で書かれているので，たとえばUNIX系OSあるいはCygwin上で下記のコマンドを実行することによってテキスト形式に変換できます．

groff -mdoc -Tascii queue.3

また，-Tasciiの代わりに-ThtmlとすればHTML形式の，-TpsとすればPostScript形式のマニュアルを得ることができます．

queue.hは，単方向リスト(SLIST)，末尾へのポインタ付き単方向リスト(STAILQ)，双方向リストの変種(LIST)，末尾へのポインタ付き双方向リストの変種(TAILQ)，および環状リストの変種(CIRCLEQ)を提供します．

tree.hは，平衡探索木の一種である赤黒木(RB)と，自己調整型探索木であるスプレー木(SPLAY)を提供しています．

BSDライセンスなので，組み込みソフトウェアでもとくに問題なく利用することができると思います．

queue.hやtree.hは，マクロで実装されていることもあって，型に関して安全なコンテナライブラリです．またqueue.hはマクロのみによる実装なので，速度的なオーバヘッドもありません．tree.hの場合，キーの比較処理は必ず関数呼び出しが入るので，インライン関数機能を利用しない場合，STLに比べて若干の速度的なペナルティがあるかもしれません．

● 侵入的データ構造の採用

queue.hやtree.hは，STLやglibとは一風変わった形で，データ構造を提供します．STLやglibの場合，コンテナに格納するそれぞれの要素データとコンテナは，まったく独立したデータ構造でした．このため，既存のデータ型をコンテナに格納する場合，既存のデータ型のほうに変更を加える必要はありません．

これに対し，queue.hやtree.hの場合，コンテナ型が必要とするリンク構造を，コンテナに格納される要素データ型の側に用意してやる必要があります．これは特集第3章Part 1の先頭で，コンテナを使わない悪い例として言及されている方法そのものです．

たとえば，ある構造体が，一つのSLIST型コンテナ，二つのTAILQ型コンテナ，一つのRB型コンテナの計四つのコンテナに同時に格納される場合，その構造体がメンバ変数として，一つのSLIST_ENTRY，二つのTAILQ_ENTRY，一つのRB_ENTRYを含むように定義してやる必要があります．

このように，コンテナ型に格納される要素データ型に，追加データが必要となるようなデータ構造を，内生的(endogenous)

あるいは侵入的(intrusive)なデータ構造と呼びます．これに対し，STLやglibのように，要素データ型に変更を加える必要のないデータ構造を，外生的(exogenous)あるいは非侵入的(non-intrusive)なデータ構造と呼びます．C++言語の文脈では，侵入的/非侵入的といういい方をすることが多いようです．

では，なぜqueue.hやtree.hでは，あえて再利用性などの点で都合の悪い侵入的なデータ構造を選んでいるのでしょうか．

第一の理由は，データ要素へのポインタを元にO(1)のコストでその要素を削除したり，その要素の前後に他の要素を挿入できるからです．これに対し，非侵入的なglibのg_listでは，要素を指すgpointerを元にg_list_remove()を行うとO(*N*)のコストがかかります．ある構造体が複数のリストに格納されている際に，その構造体をすべてのリストから削除する操作は，侵入的構造により効率が改善できる典型的な例です．

第二の理由は，動的なメモリ確保を避けることができるからです．非侵入的なデータ構造を採った場合，コンテナに要素を追加するたびに，_ENTRYに対応するデータを動的にメモリ確保する必要があります．侵入的なデータ構造の場合，要素データ型がすでに_ENTRYを含んでいるわけなので，動的メモリ確保も不要で，また，それにともなう速度的オーバヘッドもありません．各要素データを静的な配列に確保しているような場合，侵入的なデータ構造を用いれば，動的なメモリ確保を一切なしに，この配列要素をリストや探索木に格納することができます．

しかし，もちろん第3章で述べられているように，侵入的データ構造には，問題もあります．上で述べたような理由の当てはまらない場合には，queue.hやtree.hを使って非侵入型のデータ構造を実現することもできます．それには，要素データ型としてポインタ型のメンバー一つだけを持つ構造体を用い，真の要素データは，そのポインタの指す先にとるようにすれば良いのです．

● queue.h

各種のリスト操作マクロを提供します．

リスト9は，glibのg_slist，g_list，g_queueの例と同様に，入力した行をリストの先頭につけ加え，逆順で表示する例です．この例ではTAILQ(末尾へのポインタ付き双方向リストの変種)を使っています．リスト9の先頭のstruct stringの定義では，侵入的データ構造の項で説明したとおり，要素データ型であるstruct stringに，リストのリンク構造のためのメンバ変数tailq_linkを追加しています．tailq_linkの前のTAILQ_ENTRY(string)の部分は，実際には次のように展開されます．

```
struct {
    struct string *tqe_next;
    struct string **tqe_prev;
}
```

このようにリンク構造を定義するマクロの引き数には，リンクが指す要素データ型のstruct名称(この場合はstring)を渡します．

〔リスト9〕bsd_tailq.c

```
#include <stdio.h>
#include "subr.h"

#include <sys/queue.h>

struct string {
    TAILQ_ENTRY(string) tailq_link;
    char *s;
};

TAILQ_HEAD(string_list, string);

int
main()
{
    struct string_list list;
    struct string *node;
    char *line;

    TAILQ_INIT(&list);
    while ((line = get_line_alloc(stdin)) != NULL) {
        node = emalloc(sizeof(*node));
        node->s = line;
        TAILQ_INSERT_HEAD(&list, node, tailq_link);
    }
    TAILQ_FOREACH(node, &list, tailq_link) {
        printf("%s¥n", node->s);
    }
    return 0;
}
```

次のTAILQ_HEAD()は，struct stringのリストを表す型struct string_listの定義です．この部分は，実際には下記に展開されます．

```
struct string_list {
    struct string *tqh_first;
    struct string **tqh_last;
};
```

もしstruct stringやstruct string_listを他のソースファイルでも利用するのであれば，ここまでの部分は，ヘッダファイル中に記述することになります．

main()関数の先頭のTAILQ_INIT()では，listを空にしています．変数の初期化の場合には，次のようにして空にする方法もあります．

```
static struct string_list list =
                TAILQ_HEAD_INITIALIZER(list);
```

whileループの中のTAILQ_INSERT_HEAD()では，リストの先頭にnodeを挿入しています．このTAILQ_INSERT_HEAD()マクロには，リストを表す変数list，リスト要素を表すnode以外に，リスト要素であるstruct stringのメンバ変数tailq_linkの名称を与える必要があることに注意してください．これは，リスト要素が複数のTAILQ_ENTRYを含む場合に，どちらのTAILQ_ENTRYをたどるのかを指定するために必要なのです．

表示処理の部分では，TAILQ_FOREACH()を用いてTAILQをすべてたどっています．ここでメンバ変数tailq_linkの名称をマクロの引き数に与えている理由も同じです．

このTAILQ_FOREACH()は次のように書き下すことも可能です．

```
for (node = TAILQ_FIRST(&list); node != NULL;
     node = TAILQ_NEXT(node, tailq_link))
```

このTAILQ_NEXT()は実際には次のように展開されます．

```
(node)->tailq_link.tqe_next
```

tailq_linkという名称が，マクロの引き数として必要な理由がわかると思います．

queue.hには多種類のリスト構造が定義されていますが，提供している機能の違いはあるものの，各マクロの引き数や使い方はほぼ同じです．このため，たとえばリスト9のTAILQをSLISTに書き換えるだけで，利用するデータ構造を単方向リストに変更することができます．

ただし，CIRCLEQだけは注意が必要で，リストの末尾かどうかの検査を，

```
node != NULL
```

ではなく，

```
node != (void *)&list
```

のように行わなければなりません．各リスト構造の提供する機能を表1に示します．表でO(N)と記されているのは，マクロは提供されているものの，リストの先頭からの要素数に比例する実行時間のかかる機能です．この機能を必要とする場合には，同じ機能をO(1)で実現できる，他の種類のリストを利用したほうが良いでしょう．

queue.hにあるリスト構造のうち，STAILQ，TAILQ，CIRCLEQはリストの末尾へのポインタを保持しているため，リスト9のTAILQ_INSERT_HEAD()の部分を，

```
TAILQ_INSERT_TAIL(&list, node, tailq_link);
```

のように書き換えるだけで，ファイルの内容を正順で表示するようになります．

なお，表1でLISTとTAILQに「変種」と記載があるのは，これらは通常の双方向リストと若干構造が違うからです．LISTやTAILQの場合，逆方向のリンクは直前の要素ではなく，直前の要素中の次要素へのポインタを指しています．このため，TAILQ_PREV()の実装はかなり無理をしており，またtqh_lastとtqe_prevが，同じ構造体オフセットを持つという事実に依存したコードになっています．

また，CIRCLEQに「変種」とあるのは，環状リストの先頭/末尾部分にCIRCLEQ_HEAD()型の構造体が挿入された構造になっているからです．環状リストの先頭/末尾部分の目印としては，要素と同じ型の構造体を挿入するよう方が，より一般的です．これは，そちらの方が先頭/末尾を検査する条件判断の数を大幅に減らせるためです．

● tree.h

2種類の二分探索木を提供します．

マクロ名でRBとあるのは，STLがset，multiset，map，multimapで用いているのと同じ赤黒木という種類の平衡二分探索木です．赤黒木の平衡条件は，木の根からもっとも遠い葉までのノード数が，もっとも近い葉までのノード数の2倍を越えないというものです．glibが利用しているAVL木と，tree.hの用いている赤黒木は，探索・挿入・削除の時間オーダは，いずれもO(log N)です．

マクロ名SPLAYはスプレー木と呼ばれるデータ構造です．スプレー木にはAVL木や赤黒木のような平衡条件はありません．アクセスパターンによっては，通常の二分木の場合と同様，リストと同様な構造に縮退することもあります．この縮退した状

〔表1〕各リスト構造の提供する機能

SLIST	STAILQ	LIST	TAILQ	CIRCLEQ	型 / 処理名	
単方向	単方向	双方向 変種	双方向 変種	双方向	単方向/双方向	
線形	線形	線形	線形	環状 変種	線形/環状	
	尾付		尾付	尾付	末尾へのポインタ有/無	
○	○	○	○	○	FIRST	先頭要素を得る
○	○	○	○	○	INSERT_HEAD	先頭に挿入
○	○	(a)	(a)	(a)	REMOVE_HEAD	先頭を削除
×	(c)	×	○	○	LAST	末尾要素を得る
×	○	×	○	○	INSERT_TAIL	末尾に挿入
×	×	×	(a)	(a)	(存在せず)	末尾を削除
○	○	○	○	○	INSERT_AFTER	直後に挿入
×	×	○	○	○	INSERT_BEFORE	直前に挿入
(b)	(b)	(a)	(a)	(a)	(存在せず)	直後を削除
O(N)	O(N)	○	○	○	REMOVE	削除
○	○	○	○	○	NEXT	次の要素を得る
○	○	○	○	○	FOREACH	正順に全て処理
×	×	×	○	○	PREV	前の要素を得る
×	×	×	○	○	FOREACH_REVERSE	逆順にすべて処理

(a) その名称のマクロは提供されてないが，既存のマクロを組み合せるとO(1)で実行可能．
(b) その名称のマクロは提供されてないが，実装にやや踏み込んだ記述をすれば，O(1)で実現可能．
(c) FreeBSDでは提供されているが，現在のところNetBSDにはない(おそらくソースを同期したときのミス)．

態では，当然，要素へのアクセスに$O(N)$のコストがかかることもあります．にも関わらず，スプレー木に対する何回かのアクセスを行い，ならしたコスト(amortized cost)注2を求めると，$O(\log N)$で済むというおもしろい性質をもっています．スプレー木では，アクセスされたノードを木の根に移動するというアルゴリズムで木の回転操作を行うのですが，これが木を平衡化させる効果を持つためです．また，アクセスしたノードが木の根に移動するので，LRUリストと同様なキャッシュ効果があります．このような特徴から，スプレー木は自己調整二分木(self adjusting binary tree)とも呼ばれます．

スプレー木は単なる探索の場合でも木の回転操作を行うので，その点で通常の平衡探索木よりも遅くなることがあるのですが，キャッシュ効果の効くアクセスパターンの場合などは，非常に速い探索木として使えることがあります．また，平衡化のための余分な情報が不要なので，ノードあたりのメモリが節約できる点も魅力です．ただし，木が縮退した構造になることがあるので，再帰的に木をたどるような操作が必要な場合は，スプレー木の利用は避けたほうが良いかもしれません．要素の数だけ再帰を繰り返し，スタックオーバフローを起こす可能性があるからです．もっとも，tree.hで提供されているマクロは再帰を一切使っていないので，マクロを利用している限りはその心配はありません．

リスト10は，glibのg_hash_table，g_treeの例と同様な，単語の出現頻度を数えるプログラム例です．先頭のstruct word_frequencyの定義では，侵入的データ構造の項で説明したとおり，要素データ型の中に，木のノード間リンクのためのメンバ変数rb_linkを追加しています．rb_linkの前のRB_ENTRY()は実際には次のように展開されます．

```
struct {
    struct word_frequency *rbe_left;
    struct word_frequency *rbe_right;
    struct word_frequency *rbe_parent;
    int     rbe_color;
}
```

このため，RB_ENTRY()のマクロ引き数には，リンクが指す要素データ型のstruct名称である，word_frequencyを渡す必要があります．

次のRB_HEAD()は，木の根を表す型(この場合はstruct word_map)の定義で，次のように展開されます．

```
struct word_map
        { struct word_frequency *rbh_root;};
```

ここまでは，queue.hの場合と良く似ています．

次にword_compare()関数のプロトタイプ宣言がありますが，これは連想探索キーの比較関数です．glibのg_treeの場合と同様，キーの大小関係を比較し，負の数・零・正の数の返り値で，それぞれ小・等価・大を表します．

queue.hと大きく異なるのはこの次です．tree.hの場合，マクロ展開によっていくつかの関数を定義します．RB_PROTOTYPE()は，その関数群のプロトタイプ宣言を生成するマクロです．もし，struct word_frequencyやstructword_mapを他のソースファイルでも利用するのであれば，ここまでの部分を，ヘッダファイル中に記述することになります．

次に連想探索キーの比較関数word_compare()の定義が続きます．その次のRB_GENERATE()が，関数定義を生成するマクロの呼び出しです．テンプレートの場合ならインスタンス生成に対応する処理だと考えれば良いでしょう．このRB_GENERATE()は，word_map_RB_で始まるいくつかの関数定義に展開されます．

ここまでで準備が完了したので，あとは利用するだけです．リスト10のmap_init()にあるように，RB_INIT()で木を空にします．変数の初期化の場合には，次のようにして空にする方法もあります．

```
struct word_map map = RB_INITIALIZER(map);
```

木へのデータの追加はword_add()にあるようにRB_INSERT()で行います．queue.hの場合，マクロの引き数として_ENTRY()メンバ変数の名称を渡しましたが，tree.hの場合には，ここにあるように木の根の型名(この場合はword_map)をマクロの引き数に渡します．木のノードに複数の_ENTRY()メンバ変数がある場合，使われるのは，この木の根の型名を第1引き数にしてRB_PROTOTYPE()およびRB_GENERATE()を呼んだ際に，第3引き数で指定したメンバ変数となります．

要素へのアクセスは，word_lookup()にあるようにRB_FIND()で行います．これにも木の根の型名であるword_mapを渡しています．

リスト10のRB_をすべてSPLAY_と書き換えれば，スプレー木を使うことができます．

連想探索キーの比較関数word_compare()に渡されるのは，比較に用いられるキーではなく，木のノードへのポインタですので，tree.hの場合は，glibのハッシュ表や平衡探索木の場合と異なり，構造体word_frequencyに連想キーをメンバ変数として含むことが必須になります．

C++との対応

さて，これでひととおりの紹介が終りました．最後にC++ STLとの対応をまとめておきます．

● vector

STLのvectorの実体は，動的に確保された配列です．glibのg_array，g_ptr_array，g_byte_arrayがこれに対応します．glibを使わない場合も，realloc()を用いて，自身で配列のサイズを管理し，vector相当の機能を実現するのは，それほど難しくないでしょう．

注2：処理を複数回行う際に要する合計時間を，処理の回数で割った一回あたりのコスト．時に一回あたり長時間かかる処理があっても，全体としてならしてみれば十分に効率が良いことを示すために使われる．

〔リスト 10〕bsd_rbtree.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "subr.h"

#include <sys/tree.h>

struct word_frequency {
    RB_ENTRY(word_frequency) rb_link;
    char *word;
    int count;
};

RB_HEAD(word_map, word_frequency);

int word_compare(struct word_frequency *,
                 struct word_frequency *);

RB_PROTOTYPE(word_map, word_frequency, rb_link, word_compare)

int
word_compare(struct word_frequency *a, struct word_frequency *b)
{
    return strcmp(a->word, b->word);
}

RB_GENERATE(word_map, word_frequency, rb_link, word_compare)

struct word_map map;

void
map_init(void)
{
    RB_INIT(&map);
}

void
word_add(char *word)
{
    struct word_frequency *node= emalloc(sizeof(*node)), *found;

    node->word = word;
    node->count = 1;
    found = RB_INSERT(word_map, &map, node);
    if (found == NULL) /* inserted */
        return;
    /* already exists */
    free(node->word);
    free(node);
    found->count++;
}

void
word_list(void)
{
    struct word_frequency *node;

    RB_FOREACH(node, word_map, &map) {
        printf("%d¥t%s¥n", node->count, node->word);
    }
}

void
word_lookup(char *word)
{
    struct word_frequency *node, tmp;

    tmp.word = word;
    node = RB_FIND(word_map, &map, &tmp);
    if (node == NULL) {
        fprintf(stderr,"%s: not found¥n", word);
    } else {
        printf("%d¥t%s¥n", node->count, node->word);
    }
}
```

● deque

STLのdequeの実体も，動的に確保された配列です．これに直接対応する型は，今回紹介した中にはありませんでした．しかし，realloc()を用いて自身で配列のサイズを管理し，また先頭要素と末尾要素の添字を保持することによって，deque相当の機能を実現するのはそれほど難しくありません．また，dequeのランダムアクセス機能が不要な応用であれば，リスト構造を使うこともできるでしょう．

● list

STLのlistの実体は双方向リストです．今回紹介した中ではglibのg_queueやqueue.hのTAILQが，これに近い機能を提供しています．

ただし，つねに無条件でg_queueやTAILQを使うのではなく，必要な機能を提供するより簡単なデータ構造(STAILQやSLISTなど)を選んで使うのが良いと思います．筆者の場合，リスト処理は，サブルーチンやマクロを使わずについ毎回書き下してしまいがちなのですが，多人数のプロジェクトでは，glibやqueue.hのリスト機能を使うことによって，ソースの可読性を上げることができると思います．

● set，multiset，map，multimap

STLにおけるこれらの実体は平衡二分木でした．glibのg_tree，tree.hのRBなどがこれらに対応します．各要素をsortして列挙するという機能が不要であれば，glibのg_hash_tableなどのハッシュ表を使うこともできます．性能の点からは通常はハッシュ表のほうがお勧めです．

● メモリ管理

C++ではshared_ptrなどを使った参照カウント方式のメモリ管理がよく用いられています．これは，参照回数が増えたらカウンタを増やし，参照回数が減ったらカウンタを減らすだけの処理なので，C言語でも実装自体は簡単です．実際，glibパッケージに含まれるlibgobjectライブラリには，参照カウンタ式のオブジェクト管理機能が含まれています．

しかし，Cの場合，カウンタ管理のし忘れを防ぐことが難しいので，あまり使い勝手は良くありません．そこで複雑なデータ構造を扱う場合には，boehm GCのようなconservative garbage collectorライブラリを利用することを考えてもよいかもしれません．boehm GCは，プロセスのメモリをスキャンして，オブジェクトへのアドレス参照がなくなったらメモリを回収するので，参照カウントの保守が不要であり，利用方法もずっと簡単です．基本的には#include <gc.h>を行い，malloc()をGC_MALLOC()に，またrealloc()をGC_REALLOC()に書き換えるだけで，メモリの解放を自動的に行ってくれます．詳しくは次のwebページを参照してください．

http://www.hpl.hp.com/personal/Hans_Boehm/gc/

そだ・のりゆき　(株)SRA

Appendix

Windows 環境におけるテンプレートの現在

マイクロソフトの STL サポート状況

中山 宏之

効率的な C++プログラミングを行ううえで有用な STL(Standard Template Library)技術は，マイクロソフトの開発環境でもサポートされている．ここではマイクロソフト製品での STL サポート状況を紹介する．

Visual Studio での STL サポート

マイクロソフト社のデスクトップ向け開発環境は Visual Studioと呼ばれており，統一された IDE 環境と，その中で起動可能な Visual C++，Visual Basic，Visual C# などの各種プログラミング言語のサポートからなる．Visual C++での STL サポートは，Visual C++4.2 の頃に開始され，Visual Studio 5.0，6.0，.NET と継続されてきた．もちろん最新の開発環境 Visual Studio.NET 2003に付属する Visual C++コンパイラでも STL はサポートされている．

Visual C++では STL 用ヘッダファイルは C++Standard Library の一部として提供されており，単に `#include <algorithm>` などとして使用することができる．STL コード自体は Dinkumware 社(`http://www.dinkumware.com/`，図A)のものを利用しているようで，コード中には PJ.Plauger 氏や Hewlett-Packard 社の版権表示が入っているものもある．

マイクロソフト社による C++用 STL の公式サポートの意味は，マイクロソフト社の C++コンパイラを使用して STL を利用したプログラムがコンパイル可能であるということと，コンパイルされたコードの動作が期待どおりのものであることを保証するということである．コンパイラ開発の過程で(どの程度深く かはわからないが)コンパイルテストとコンパイル結果の動作検証が行われたことを示している．

Visual Studioのマニュアル中には STL のリファレンス資料[注1] が存在し，英語ではあるが各テンプレートの使用法が説明されている．ただし，(実際に読んでみたが)その内容はやはり難しく，プロジェクトで使用するには何らかの解説書の併用が必要だと感じた．

Windows CE.NET での STLサポート

Windows CE.NET の開発は，CE プラットホーム開発ツールの Platform Builder(最新版はバージョン 4.2)とそれぞれのプラットホー

〔 図A〕Dinkumware 社の Web ページ



注1：オンラインの MSDN サイトからもアクセスできる．`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcstdlib/html/vclrfCplusplusLibraryOverview.asp`

〔図B〕PocketPC2003 SDKのダウンロードページ



ム向けアプリケーションを開発するeMbedded Visual C++(最新版はバージョン4.0 SP2)を利用して行う．そして，このどちらもSTLをサポートしている．

当然ながらWindows CE.NETはx86以外にもMIPS，ARM，SHxの各種プロセッサファミリをサポートしており，このいずれのコンパイラを用いてもSTLは使用できる．また，PocketPC2003，SmartPhone2003もそのベースOSはCE.NET 4.2であるため，それぞれのSDK(図B)[注2]内にはSTL用ヘッダファイルが入っている(これ以前のPocketPCでは公式サポートされないことに注意)．

Platform BuilderのSTL用ヘッダファイルは，ビルドツリーのPUBLIC¥COMMON¥SDK¥INCに格納されている．そのためPlatform BuilderのExport SDK機能で作成するすべてのカスタムSDKにもSTL用ヘッダファイルが含まれることになる．

こちらのライブラリもDinkumware社由来のものだろう．実際にCE.NET版のSTL用ヘッダファイルをVisual Studio6.0に含まれるものと比較してみたが，記述の細部が異なるのみでほとんど同じものである．ただし，CE.NETにはiostreamsのサポートがなく，STLのみのサポートとなる．

Windows CE 3.0でもSTLは使える？

余談だが，筆者の会社では以前Windows CE 3.0で，もともとのHP社のSTLライブラリ[注3]の中からvector，deque，map，setをプロジェクトに使った経験がある．もちろんこのような使い方をマイクロソフトでは保証しているわけではない．すべて使用者の責任で検証およびテストを行って，問題はないとわかっている部分に限定して使うべきものであろう．

STLは(作るのはとってもたいへんそうだが)非常に使える便利なテクノロジである．苦労してこのようなしくみを構築してきた先人たちに感謝しつつ，プロジェクト効率化のため大いに利用すべきものであろう．

なかやま・ひろゆき　ピースクウエア(株)

注2：PocketPC2003 SDKは http://www.microsoft.com/downloads/details.aspx?displaylang=ja&FamilyID=9996B314-0364-4623-9EDE-0B5FBB133652 からダウンロードできる．

注3：現在は ftp://ftp.cs.rpi.edu/pub/stl/hp/ などからダウンロードできる．

Information ── 沖電気，CTI & IPコミュニケーションシステムのIPコミュニティセンタに二つの機能を追加
沖電気工業(株)は，CTI & IPコミュニケーションシステム「CTstage 41 for .NET」のIPコンタクトセンタにASP型コンタクトセンタ業務を支援するマルチテナント機能と，IP電話の050サービス網への接続機能を追加した．

組み込みGUI設計の現状とソリューション

第3回（最終回） iWinのActiveXコントロールによる機能拡張

中山 宏之

前回までの連載でWindows XP Embeddedをターゲットとした UIツール「iWin for XP Embedded」の組み込み方法を説明しました．連載の最後となる今回はActiveXコントロールの概念と，これを利用したiWinの拡張法を説明し，Visual Studio .NET 2003を利用してiWinで利用できるATLベースActiveXコンポーネントを作ってみます．



ActiveXコントロールとは何か

● Windowsでのコード共有

ご存知のようにWindowsが普及する以前，DOSベースで動いていた昔のパソコンはすべてシングルタスク処理だったので，DOS上でマルチタスクを意識しなければならない場面は，割り込み処理を記述する部分と，いわゆるTSR[注1]プログラムを記述するときぐらいでした．もちろん8ビット時代から（たとえばOS-9などで）マルチタスク処理は一部で可能ではあったのですが，主流のDOSではほとんど意識されませんでした．

最初のリアルモード（16ビット）Windowsが登場したときも，実行速度を上げるためか，Windowsプログラムは動作中にプログラムコード中のセグメント値を動的に変更しながら動作していました．本来，真にマルチタスクなOS（プリエンプティブなOS）ではこのようなコード書き換えは不可能です．しかし，実際には16ビットWindowsはノンプリエンプティブなOSであり，API呼び出し中にしかタスクスイッチが起こらないので問題はありませんでした[注2]．

これに対して，プリエンプティブなOS[注3]として設計されたWindows NTでは，すべての32ビットコードはどこで実行を中断されても問題のない，完全にリエントラントなコードとしてコンパイルされます．また，コード領域の動的な書き換えは不要になり，コード領域は読み込みのみ可能（Read Only）な領域として扱われるようになりました．

16ビットWindowsもWindows NTをはじめとする32ビットWindowsも，それぞれのスケジューリング方式において動作するプログラムは，一つのプログラムコードを複数のプロセスで共有することができます．たとえばノートパッドをいくつも実行したとしても，その実行コードはメモリ上に一つのコピーがあれば良いのです．

この動作はオブジェクト指向でいうクラスとインスタンスの関係に似ています．コード部分（＝クラス）は一つだけしかありませんが，プロセスで使用するデータ，スタック，その他プロセス管理用メモリを複数セットもつことにより，複数のプロセス（＝インスタンス）を利用することができるのです．

● DLLによるコードの再利用とCOM

DLL（ダイナミックリンクライブラリ）を採用すると，複数のプログラムで共通に使用するコード部分を一つのDLLにまとめることにより，実行時に必要なメモリ量をさらに削減することができます．

ところが，WindowsでのDLLの多用には問題がありました．

- 仕様が微妙に違うDLLのバージョン管理の問題
- 同じモジュール名のDLLが複数存在できないという問題
- DLLにはサポートする関数名が保存されるが，あらかじめ名前（または番号）がわからないと呼び出せない．加えてDLLには呼び出しパラメータの情報を保存できない

Windows上でこれらの問題を解決する一つの方法がCOM（Component Object Model）というしくみです．COMではインターフェースの仕様とその実装が分離され，インターフェース仕様自体をGUID[注4]で一意に識別するしくみを設けました．そのためプログラムが使用するインターフェースを（名前ではなくGUIDで）厳密に指定することができます．また，集約というしくみを使って微妙に異なる複数のインターフェースを一つのCOMでサポートすることもできます．

ここでいう「インターフェース」とは，オブジェクトを操作す

注1：Terminate and Stay Residentプログラム．動作を開始するとメモリ上に常駐終了し，別プログラム動作中にホットキーなどで呼び出せる．

注2：割り込み処理の復帰時にはレジスタなどすべての状態を元に戻す．ただし，割り込みによってタスクを中断した後，そのまま別タスクを実行することはできない．

注3：タイマ割り込みのタイミングでいつでもタスク中断が可能なOS．

注4：Globally Unique IDentifier：128ビットの情報量をもち全世界で一意（ぶつかることがない）と考えられる数．EthernetのMACアドレスと時刻情報などを組み合わせて生成される．

る一揃いの関数仕様の集まりだと思ってください．C++的には，インターフェースとは実装をともなわない(呼び出し方式のみ規定した)クラスのようなもので，オブジェクトを実際に生成するクラスは使用するインターフェースをそのまま継承します．COMでは各オブジェクトをインターフェース経由で操作することができますし，逆に新たにオブジェクトを生成するようなインターフェースを定義することもできます．

先ほど見たように，COM以前のWindowsプログラムは単純なクラス-インスタンスの関係と見られましたが，COMを使用することによってWindows上で本格的なオブジェクト指向プログラミングが可能になります．しかし，このような方法は実際にはオーバヘッドが大きいため，十分なCPUの処理能力とメモリ容量がなければ成立しません．実質的にはWindows 95，Windows NT 4.0以降，Pentium IIプロセッサが普及したころ，初めて実用的になったと考えられます．

● コンポーネントとコントロール

プログラムを再利用可能な部品として用意することにより，

- 一度開発した成果を別のプロジェクトで再利用する
- 第三者の開発した成果を(対価により)入手して利用する

などのことが可能になります．このように部品として使用可能なソフトウェアのまとまりをコンポーネントと呼びます．

一時期，DLLはこのようなソフトウェア部品でしたが，作成方法にあまりにも制約がなく，バージョン管理などの問題もあることから，最近ではDLLがそのままソフトウェア部品として流通することは特殊な場合(=プラグインなど仕様が明確な場合)をのぞき，なくなりつつあります．その代わりCOMや(後述する)ActiveXのテクノロジを利用して作成されたソフトウェア部品が使用されるようになり，COMコンポーネントやActiveXコンポーネントなどと呼ばれます．16ビット版のVisual Basicが盛んだった頃は，Visual Basicで利用可能な部品としてVBXが流通していました．一時期，32ビットVisual Basicで利用可能なソフトウェア部品はOCXであると説明されていましたが，これは実際には(ある条件を満たした)COMコンポーネントのことです．

一方，Windowsではユーザーインターフェースを提供しているソフトウェア部品のことを「コントロール」と呼びます．たとえばWindowsのいちばん基本的なGUI部品であるボタンやスクロールバーなどは「Windowsコントロール[注5]」と呼ばれ，IEとともに導入されたRebarやTreeViewなど，もう少し高度な部品は「コモンコントロール」と呼ばれます．これ以外にもHTML上で利用可能なコントロールや，.NETプログラム(WinForm)で使用可能なコントロールも用意されています．

コンポーネントはコントロールを含む概念ですが，一方でかならずしもユーザーインターフェース(ウインドウ描画)をもたなくても良いという特性があります．COMの場合にもCOMコンポーネントといったりCOMコントロールといいますが，前者の場合には(かならずしもUIをもたないが)一般的にソフトウェア部品，後者の場合にはUIをもっているソフトウェア部品というような意味になります．

● ディスパッチインターフェース，オートメーション

COMコンポーネントは本来C++と非常に親和性の高い，効率的な呼び出しのできるソフトウェア部品化の方法です．ところがこの効率的な呼び出し方法[注6]はDLLと同様に，

- 呼び出しモジュールのコンパイル時に，使用するCOMのインターフェース仕様があらかじめ(ヘッダファイルや.IDLファイルで)わかっている
- リンク時にすべてのCOM呼び出しのパラメータが一致しなければならない
- COMにどのようなインターフェースがあるかリストできない

などの欠点があります．この部分の柔軟性を増すために使用可能な方法が，ディスパッチインターフェースを使用する方法[注7]です．

ディスパッチインターフェースの特徴は，

- インターフェースの定義をリソースとしてもつ
- パラメータに使用できるデータ型が決まっている
- 動的バインディングあるいはレイトバインディングが可能
- コンポーネントの自己登録機能，自己削除機能をもつ

などです．ディスパッチインターフェースはVisual BasicやVBScript，JScriptなどのC++以外の言語をサポートするために使用され，そのパラメータ型にはVisual Basicでも利用されるVARIANT型が多用されます．このインターフェースをサポートすることによってコンポーネントの柔軟性が大幅に増しますが，その代わりコンポーネントのロード時やインターフェース呼び出し時のオーバヘッドが大きくなります．

もちろん，通常のインターフェース呼び出しとディスパッチインターフェース呼び出しを両方実装してしまうことも(ATLを使用すれば簡単に)でき，これをデュアルインターフェースと呼びます．

COMのインターフェースは実際には複数の関数定義が可能ですが，ディスパッチインターフェースではこの関数の種類を以下の3種類として(モデル化して)考えています．

- プロパティ
  COMオブジェクトの属性としてある値を設定したり，逆に値を取得できる．値を設定することによって何らかの動作を引き起こしたり，コントロールのRedraw時に値を使用する．
- メソッド
  COMオブジェクトに命令を出すことによって何らかの働き

注5：Windowsコントロールはシステムであらかじめ登録されたクラス名をもっている．作成するときはCreateWindowにそれぞれの決まったクラス名("BUTTON"など)を指定する．
注6：VTable呼び出しなどと呼んでいる．
注7：具体的にはIDispatchインターフェースを継承する．



を行わせる．戻り値があっても良い．

●イベント

COMオブジェクトから何らかの通知を受け取る．上記二者とはメッセージの方向が逆になる．

● COMサーバの三つの形式

COMのしくみはその機能を提供する「COMサーバ」と機能を使う「COMクライアント」の二つの要素から成り立っています．ここまで説明してきた部品としてのCOMコンポーネントはCOMサーバの一種と考えられます．COMサーバには以下の三つの形式があります．

●インプロセスCOM

DLLとして実装するCOMです．COMのサービスを利用する「COMクライアント」と同じプロセスにあるため，単にファンクションコールを行えばよく，インターフェースのオーバヘッドが小さくてすみます．

●アウトプロセスCOM

クライアントのプロセス外でEXEとして実装されているCOMです．WORDやEXCELなどのCOMオブジェクト（文書）を操作する場合，実際にはWORDやEXCELが裏で立ち上がっており，クライアントはCOMを介して独立した外部のプログラムに処理を依頼する[注8]ような形になります．

●Distributed COM（DCOM）

ネットワークを介して別マシンで動作しているCOMです．Windowsの世界でいちばん一般的なRPC（Remote Procedure Call）の実装だと考えてさしつかえありません．

COMの呼び出しはインプロセスCOMのもっとも単純な場合，通常のC++呼び出しと同様に「スタック」を用いてパラメータのやり取りが行えますが，インプロセスの場合でも何かパラメータに細工が必要な場合（たとえば型変換が必要とか，別スレッドへ受け渡す場合），さらにアウトプロセスやDCOMの場合には何らかの「パラメータを受け渡すための特別な方式」が必要になります．COMにはこのような受け渡しのしくみがあらかじめ用意されており，それをマーシャリングと呼びます．

マーシャリングを行うための方法がいくつか用意されていますが，その中の一つにCOMの形式的なパラメータ定義ファイルである`.IDL`ファイルを用いてマーシャリング用コードを自動生成する方法があります．このときMIDLコンパイラが使用されます．

● ActiveXコントロール

ActiveXコントロールは実際のところ概念的にあいまいな存在ですが，その実は「ディスパッチインターフェースをもつCOMコントロールのうちインターネットおよび軽量化を意識したもの」程度の意味です．

COM技術のもともとの始まりは，WORD文書の中にEXCELの表を埋め込むようなOLE（Object Linking and Embedding）として，16ビットWindowsの時代に始まりました．OLEにはもともと「プログラム間協調」の部分があり，その後OLEが32ビット化するとともに，「埋め込みサポート」の意識を切り離したものがCOMの最初のイメージです．素のCOMコンポーネントにディスパッチインターフェースを実装してコントロールにしたものが32ビットVisual Basicで使用するソフトウェア部品OCXになりました．

しかしOCXの実装は一般にプログラマの荷が重く，これを軽減するためにマイクロソフトはMFCを利用することにしました．ところが，マイクロソフトがネットスケープナビゲータのプラグインに対抗し，インターネットエクスプローラでOCXをサポートできるようにし，これをActiveXコントロールと名付けました．この時点では比較的重く，しかもMFCランタイムがなければ動作しないような方法でしかActiveXコントロールは動作しなかったのですが，その後に新しく発表されたATL（Active Templete Library）技術によって軽量のActiveXコントロールが作成可能になりました．正確にはMFCを利用した（軽量でない）ActiveXも認められていますが，ActiveXといった場合はどちらかというとコンポーネントのサイズや証明書などで，インターネットによる流通を意識したものを指すように思われます．

● iWinとActiveX

さて，ここでiWinの世界に戻ります．実際にはiWinの各コンポーネントもATLを利用して作成されたActiveXコンポーネントです．iWinコンポーネントのうちiWinShellとiWin Browserは同時にスクリプトホストの機能も果たしており，そ

Column

スクリプトエンジンを再利用する

Windowsのスクリプトサポートは IE（インターネットエクスプローラ）のスクリプト実装から始まりました．マイクロソフトの開発者たちはスクリプトエンジンをIEだけのものではなくWindows OSのコンポーネントとして開発しました．また，当時対応を迫られたマルチメディアファイルについてもOCX（当時はOLE98と呼んでいた）を利用することとし，これらすべてをCOMで構築しました．

スクリプトの実行は，

●スクリプトエンジン（VBScript，JScriptほか）

●スクリプトホスト（エンジンを利用するプログラム）

の両者が協調することで動作し，デバッガ対応を含めた実装方法もMSDNのWebサイトで公開されています．詳しくは，`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/scripting.asp`などを参照してください．

注8：これをオートメーションと呼ぶ．オートメーションは，じつはIDispatchと関係がある．

れぞれグローバルスクリプトとローカルスクリプトをホストします(コラム参照).

iWinのコンポーネントはもっぱらJScriptによってコントロールされるため，もちろんディスパッチインターフェースもサポートしています．

また，iWinBrowserの生成するブラウザUI画面の中に，Windows Media Player OCXやMacromedia Flash, Office File Viewerのようなサードパーティの ActiveXコントロールや，自分で作成したActiveXコントロールを埋め込むことももちろん可能です．このようなActiveXコントロールを利用する動機として次のようなことがあげられます．

▶スクリプトでは直接実行できないWin32 API呼び出しやドライバ操作などをActiveXコントロールで実現する．

たとえば，FAX送信を行ったりロボットアームを操作するようなActiveXコントロールを使用することが考えられます．テキスト描画とグラフィクファイル表示以外の複雑な画面描画(たとえばグラフ表示など)もスクリプトから直接実行することはできないので，それらの機能をもったActiveXコントロールを使用します．

▶スクリプトで直接実行すると性能の問題が発生する．

たとえば，高度な演算処理やグラフィク処理を行うようなActiveXコントロールを使用することが考えられます．

このとき，これらのActiveXコントロールはディスパッチインターフェースをもち，iWinBrowserのローカルスクリプトやアプリケーションHTMLに埋め込まれたスクリプトによって，オートメーション動作が可能になっている必要があります．



ATLを使ったActiveXコンポーネントの作成

● ATLとは

以前のWindowsプログラマにとって，ActiveXコンポーネントの作成は悪夢に近いものでした．当時やっとMFCを使うことによってOCXが多少作りやすくなったかな???(プログラムサイズは増えたが…)といった状況だったのですが，ATLが利用できるようになって世界は一変しました．

ATL(Active Templete Library)はMFCで実現しようとしていたWindows APIのクラスライブラリ化という命題を，もう一度別の実現方法で構築し直したものと考えられます．たとえばMFCでウインドウを扱うための一般的なクラス`CWindow`や`CDialog`はほぼ同等のものがATLでも用意されました．ただしATLの真髄はC++のテンプレート機能を利用して多数のCOM関係の機能を整備したことでしょう．このテンプレートライブラリとVisual Studio IDE[注9]に統合されたWizard機能により，あらかじめ用意されたクラスを継承し,(カスタマイズされ)作成されたオブジェクトのプログラミングがたいへん容易になりました．テンプレート機能とC++の多重継承がなければ，ウイザードの実装はもっとずっとたいへんになったはずです．

ともあれ，ATLはウイザードと組み合わせることによって，軽量なActiveXコンポーネントの作成を容易にサポートできるようになりました．この項では簡単なActiveXの作成法を紹介することによって，ActiveXコンポーネントの「プロパティ」，「メソッド」,「イベント」がどのように作成され，テストされるのかを説明します．

● Visual Studio .NET 2003の紹介

さて，本稿でATLを利用してActiveXの作り方を紹介するにあたり，どの開発環境を利用するのが良いかを考えました．じつは開発環境を考える前に動作させるターゲット環境を決める必要があるのですが，今回はWindows XP Embeddedで動かすことをメインとして，デスクトップのWindows XPとWindows 2000までをターゲットとして考えます．そして，開発環境として，最新のマイクロソフト Visual Studio .NET 2003(おもにProfessional版)を試すことにします．

Visual Studio .NET 2003の最大の特徴は，もちろん.NET Frameworkに対応したアプリケーションの開発ができるところです．そのためVisual Basic .NET，Visual C# .NET，そしてVisual J# .NETで開発が可能です．これ以外にもWebサーバ側の開発(ASP.NET)やデータベースの開発(ADO.NET)などの機能があります．Professional版以上のシステムではスマートデバイス(要するにPocket PC/SmartPhone/Windows CE .NET)をターゲットとしたVB.NET，C#での開発も可能です．

では，MFCやATLを使うようなこれまでのC++の開発はできないのか，といえばそのようなことはなく，むしろ(あたりまえだが)Visual Studio 6.0の頃とくらべても機能が向上し，またIDEも使いやすくなっているように思われます．ということで，いくつかの難点はある[注10]もののVisual Studio .NET 2003(Professional)を利用することにしました[注11]．

● 開発環境を用意する

マイクロソフトの開発環境を利用する場合，以前からWindows NT系のOSを利用することが推奨されています．したがって，OSとしてWindows 2000 ProfessionalかWindows XP Professional[注12]を搭載したPentium III 1GHz以上，メモリ

注9：IDEとはGUIを利用した統合開発環境のこと．この中にはコンパイラ，デバッガ，コードエディタ，プロジェクト管理およびマニュアル参照などの機能が含まれる．

注10：Windows 2000やXPはVisual Studio .NET 2003用ランタイムライブラリが標準で含まれない．Visual Studio 6.0で開発した場合は，たまたま2000やXPには標準で必要なランタイムライブラリが含まれているので楽ができた．

注11：たまたま入手できたのが英語版だったこともあり，以下の説明は英語版で行う．

注12：マイクロソフトの開発ツールの場合，XP Homeでは動作テストを行っていない可能性がある．もちろん動かないはずはないのだが，トラブルに遭遇した場合には苦労するかもしれない．

256Mバイト以上を搭載したマシンを用意したいところです．またサンプルで使用する関係上，開発機にはCD/DVD-ROMドライブが装備され，音楽CDの再生が可能になっているものを選んでほしいと思います．

Visual Studio .NETのインストールは以下の四つの手順に分かれています．

▶インストールのための準備：ここで無条件に.NET Frameworkのインストールが行われる．Webサービスを開発するにはIIS（Internet Information Server/Service），要するにWebサーバ機能が必要といわれるが，今回はインストールする必要はない．

▶Visual Studio .NETのインストール：ここでは“Language Tools”から“Visual C++ .NET”と“Visual C# .NET”のみインストールしてみた．

▶ドキュメントのインストール：必要に応じてMSDNライブラリをインストールする．CDから参照するようにもできるが，よく使う部分はハードディスク内にないと不便．

▶サービスリリースのチェック：もしVisual Studioのサービスリリースが存在する場合，ここでインストールすることができる．またWindows Updateを行うように勧められるのでUpdateしておく．

上記の環境においてはインストール中に再起動を要求されます．インストールが済んだらOSのバックアップを作成しておくのも良い考えです．

● サンプルプロジェクトのビルド

開発環境がうまく機能するかどうかを確かめるために，あらかじめ用意してあるサンプルプロジェクトをビルドしてみましょう．ここではATLコントロールのサンプルとして用意してある`CDInfo`というプロジェクト（Visual Studio .NET用語では「ソリューション」）を使うことにします．これはMCIコマンドを利用してCD-ROMドライブに挿入された音楽CDのトラック情報を画面にグラフィカルに表示し，マウスクリックで選択されたトラックから再生を開始するというものです．

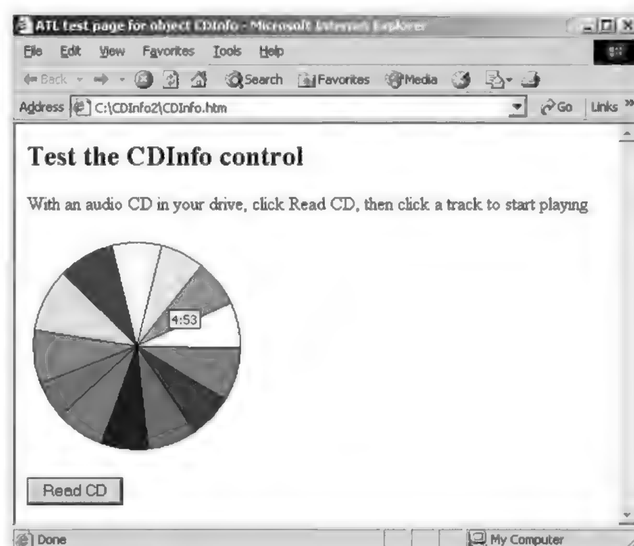
スタートメニューから“Microsoft Visual Studio .NET 2003”を選択して起動するとVisual Studio .NETのIDEが立ち上がります．“Help”メニューから“Search”を選んで表示されるサーチペインに“CDInfo”と入力して検索を開始すると，“CDINFO Sample”と“CDINFO Attributes Sample”の二つが見つかるので，ここでは前者をダブルクリックします．するとCDINFOサンプルで使用するファイルが表示されるので，“Copy All Files”をクリックし，適当なフォルダにコピーします[注13]．

コピーが終わったら，改めて“File”メニューの“Open Solution...”を実行し，コピー先フォルダの“`CDIndo.sln`”を開きます．“Solution Configurations”のドロップダウンからプロジェクトの種類として“Release”を選択し，“Build”メニューから“Build Solution”を選びます．うまくいくとOutputウインドウに，

```
Build: 1 succeed, 0 failed, 0 skipped
```

と表示され，ソリューションのビルドが完成します．このプロジェクトはATLで作成したActiveXコントロールですが，じつはビルドが終了した時点でコンポーネントとしてシステムに登録済みになっています．あとは何らかのツール[注14]で動作の確認を行えばよいのですが，プロジェクトには動作確認用の“`CDInfo.htm`”というHTMLファイルが含まれています．これをエクスプローラで開くと，図1のような画面が表示されます．音楽CDを開発機のCDドライブに挿入し[注15]，「Read CD」ボタンを押すと，今作成したActiveXコントロールに対してReadメソッドの呼び出しを行い，CDトラックの情報をコントロールに表示します[注16]．あとは好きなCDトラックをクリックすればそのトラックから音楽再生が始まります．

〔図1〕CDInfoの画面



● CDInfoのActiveX仕様

前節では単にCDInfoサンプルをビルドしただけだったのですが，じつはこのようなActiveXコントロールはVisual Studio .NETのウィザード機能によって簡単に作成できます．そこで，ここからはまずCDInfoサンプルがActiveXとしてどのような仕様をサポートしているのかを調べ，それと同じインターフェー

注13：ちなみにMSDNのWebサイトからもダウンロード可能．
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample/html/_sample_atl_cdinfo.asp`
注14：たとえば，“Tools”メニューの“ActiveX Control Test Container”などが使用できる．
注15：ここで音楽CDに対するアクションを聞かれるかもしれないが，キャンセルする．
注16：このとき，IEのセキュリティ設定によっては“ActiveXコントロールを実行しますか”のようなダイアログが表示されるので“Yes”を押す．

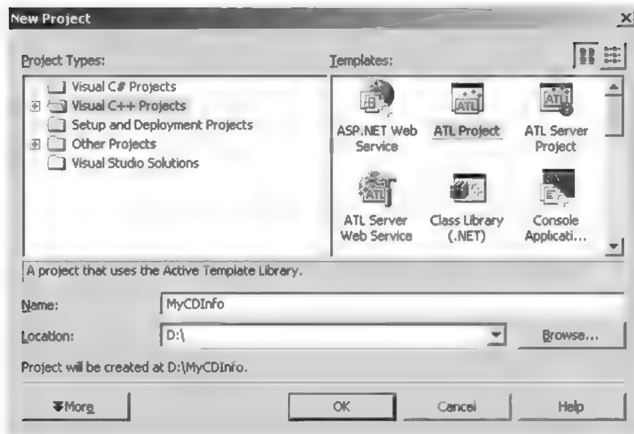
〔図2〕CDInfoのサポートするプロパティとメソッド

```
interface ICDInfo : IDispatch
{
   import "oaidl.idl";
   import "ocidl.idl";
   [propget, id(0)] HRESULT Tracks([out, retval] short* pTracks);
   [propput, id(0)] HRESULT Tracks([in] short Tracks);
   [propget, id(1)] HRESULT Length([in] short Track, [out, retval] short* pLength);
   [propput, id(1)] HRESULT Length([in] short Track, [in] short nLength);
   [propget, id(2)] HRESULT TotalLength([out, retval] short* pLength);
   HRESULT  Read([out, retval] short* pTracks);
   HRESULT Redraw();
   HRESULT Play([in] short Track);
};
```

〔図3〕CDInfoのサポートするイベント

```
dispinterface CDEvents
{
   properties:
   methods:
   [id(1)] void Click([in] short nTrack);
};
```

〔図4〕新規ATLプロジェクト



スをVisual Studio .NETでどのように作成すればよいのかを，順を追って説明することにしましょう注17．

先ほどCDInfoのファイルを保存したフォルダから“CDInfo.idl”を探してテキストエディタで開いて見てみます．図2はCDInfoのプロパティとメソッド，図3はCDInfoのサポートするイベントを示しています．

インターフェースICDInfoは八つの呼び出し可能な（C++の意味での）関数をサポートしています．最初の二つの関数が“Tracks”という名前の読み書き可能プロパティを，次の二つの関数が“Length”という読み書き可能プロパティを，次の関数が“TotalLength”という読み込みのみ可能なプロパティをサポートすることを示しています．“Read”，“Redraw”，“Play”はそれぞれメソッドを示しています．パラメータで“retval”とあるのは，関数呼び出し時に値を返すもので，通常であれば`retval = func(arg1, arg2)`となるものが，`func(arg1, arg2, &retval)`のように変形されているものです．ここで重要なのはそれぞれの関数名で，引き数のところで使用される名前は単にプレースホルダとして存在しています．

一方，イベントのほうは`void Click(short nTrack)`という形の外部関数をこのActiveXコントロールが呼び出すことを示しています．

● ウイザードでActiveXコントロールのプロジェクトを作る

以上の情報を活用しながらCDInfoと同じ仕様のActiveXコントロールを最初からATLで作成します．Visual Studio .NET IDEでまず“File”メニューの“Close Solution”を実行し，以前のソリューションを閉じます．そして“`File`”メニューから“`New`”，“Project...”を選択します．いくつかプロジェクトタイプが表示されますが，“Visual C++ Projects”から“ATL Project”を選択し，“Name”項目にコントロールの名前（たとえば“MyCDInfo”など）を入力します．Locationは“`D:¥`”などわかりやすいところが良いかもしれません．この組み合わせで“`OK`”を押すと，プロジェクトのファイルは`D:¥MyCDInfo`に格納されることになります（図4）．

次にATL Project Wizardが立ち上がります．ここでは，“Application Settings”のページを開いて“Attributed”のチェックをはずし注18，“Allow merging of proxy/stub code”にチェックを入れた後，Finishを押します．サーバタイプはDLLとなります．

● プロジェクトにATL Controlクラスを挿入

一つのActiveXコントロールは実際には複数のインターフェースをサポートすることが可能です．ATLウィザードが生成したATLプロジェクトは生成直後にはインターフェースをサポートするクラスが一つも含まれていません．したがって，

注17：これから説明する同様の内容がドキュメント中のATL TutorialやWebでも説明されている．
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_atl_ATL_Tutorial.asp
注18：Attributeとは Visual Studio .NET以降導入された，C++の関数型としていろいろな属性を設定できる機能．これを使うと.OIDファイルなどが不要になるが，逆にC++としてのソースコードの移植性が低くなる．

次はこのようなクラスをプロジェクトに追加します．

“Project”メニューの“Add Class...”で開いたダイアログから“Visual C++”，“ATL”カテゴリ中の“ATL Control”のアイコンを選択し，“Open”をクリックします．ATL Control Wizardの画面が表示されるので，C++の“Short Name”のところに“MyCDInfo”とでも入力しましょう．そうするとほかの七つの項目に自動的に名前が入力されますが，“.h file”と“.cpp file”の項目は既存のファイル名とぶつかってしまうので，それぞれ“`MyCDInfo1.h`”，“`MyCDInfo1.cpp`”のようにファイル名をちょっと変えてあげます(図5)．

次に“Options”のページへ移動し，“Supports:”の“Connection points”の項目のチェックを入れます．これはイベントのサポートを行うために必要です．

“Finish”ボタンを押すと`IMyCDInfo`クラス(インターフェース)がプロジェクトに追加されました．

● インターフェースにプロパティとメソッドを追加

右側のClass View表示注19でIMyCDInfoをダブルクリックすると，IMyCDInfoのidl定義が表示されます．現在のところIMyCDInfoインターフェースにはプロパティ，メソッドとも存在していないので，これらを追加することにします．

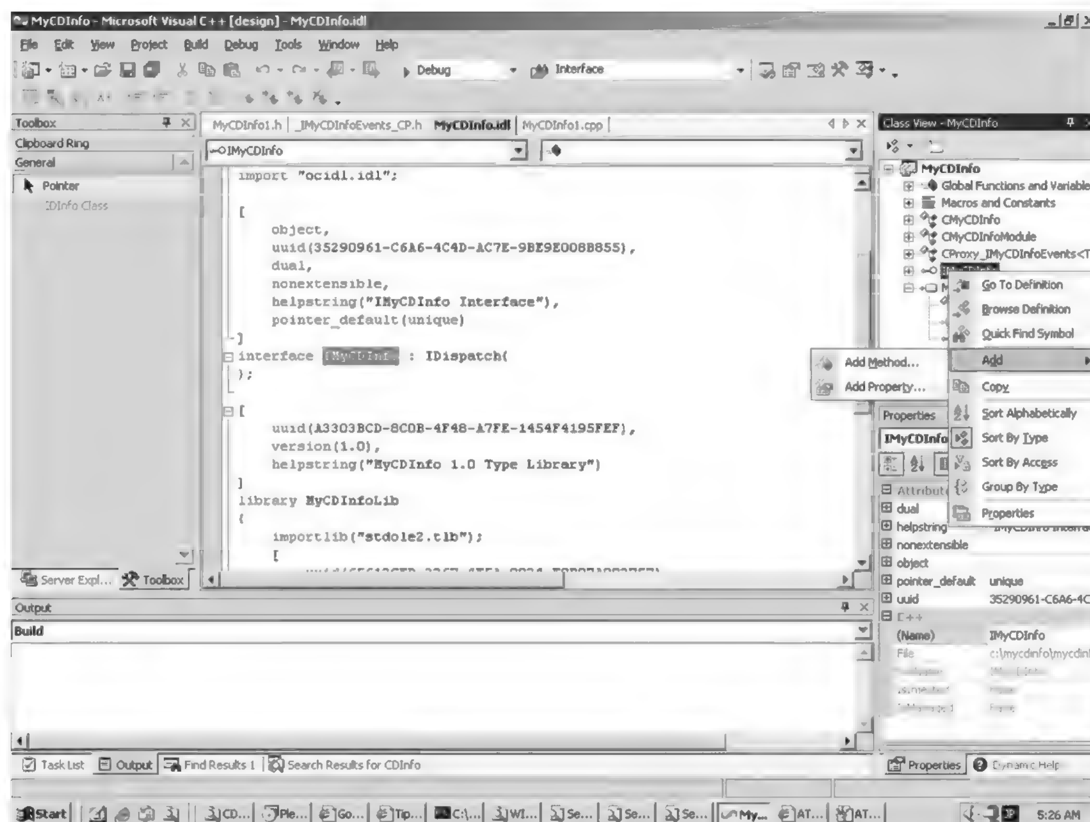
クラスビューのIMyCDInfo表示を右クリックして“`Add`”，“Add Property...”を選択してください(図6)．表示される“Add Property Wizard”で“Property type”としてSHORT，“Property name”として“Tracks”を入力します．また“IDL Attributes”のページでは“`id`”として0を入力し“Finish”ボタンを押します．これでもとのCDInfoと同様にTracksプロパティが作成されました．Lengthプロパティの場合には入力パラメータがあるので，“Parameter type”に“`SHORT`”，“Parameter name”に“`Track`”を入力してから“in”をチェックし，“Add”ボタンを押

〔図5〕ATL Controlウィザード

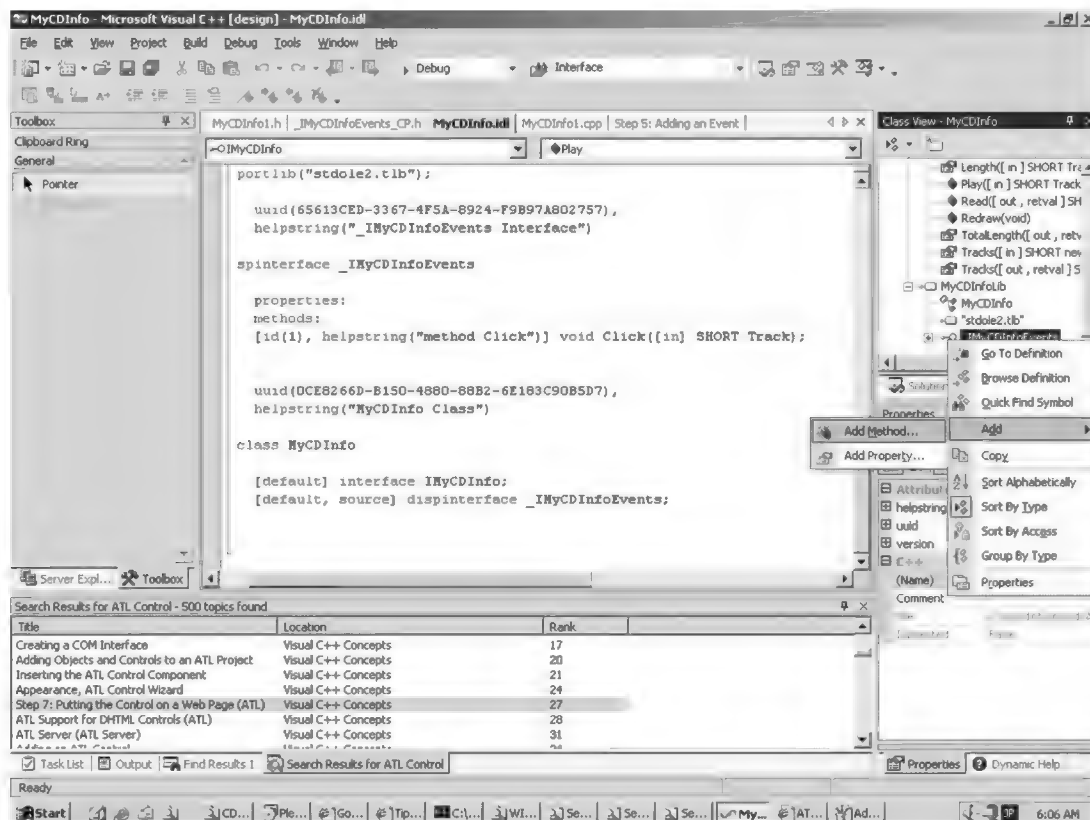


注19：Class Viewが表示されていないときは“View”メニューから“Class View”を選択する．

〔図6〕プロパティとメソッドの追加



〔図7〕
イベント呼び出し用
メソッドの追加



します．“IDL Attributes”ページでは“id”として1を使用します．“TotalLength”の場合には“Put function”のチェックをはずして同様にプロパティを作成します．要するにIDLファイルのパラメータのうちretvalと書いてあるパラメータを除いたものを設定します．

メソッドを作成する場合にも同じようにクラスビューのIMyCDInfo表示を右クリックして“Add”，“Add Method...”を選択します．メソッドの場合はすべてのパラメータを入力します．最初は“out”，“retval”が選択できないようになっていますが，“Parameter type”，“Parameter name”を入力すると変更できるようになります．この方法で“Read”，“Redraw”，“Play”のすべてのメソッドを作成します．

実際にはそれぞれのプロパティメソッドの処理内容をMyCDInfo1.cpp中のCMyCDInfo::get_Tracksなどに実装する必要があるのですが，ここでは省略します．オリジナルのCDInfoの処理内容をコピーしてきてもよいでしょう．

● プロジェクトにイベントを追加

イベントの追加は3段階で行います．まずコントロールがイベントを呼び出したいときに使用する専用のメソッドを作成します．

Class ViewよりMyCDInfoLibを探し，その下の_IMyCDInfoEventsインターフェースのアイコンを右クリックし，“Add”，“Add Method...”を選択します．最初にアンダーバーがついているインターフェースはコントロール内でローカルに使用され，外部には公開されないインターフェースであることを意味します(図7)．

“Add Method Wizard”内で“Return type”は“void”，“Method name”は“Click”とし，一つのinパラメータSHORT Trackを指定してください．“IDL Attributes”画面では“id”を1に設定して“Finish”ボタンを押します．

次に，現状の.IDLファイルからいったん.TLBファイルを生成します．まず“Solution Configurations”でReleaseを選びましょう．次に“Solution Explorer”ペインの“Source Files”項目から“MyCDInfo.idl”ファイルを右クリックし，“Compile”を実行します．すると，プロジェクトフォルダ内の一時フォルダ(DebugまたはRelease)にMyCDInfo.tlbファイルが作成されます．

この状態でIConnectionPointインターフェースを継承したコネクションポイントを作成します．Class View画面でCMyCDInfoクラスアイコンを右クリックし，“Add”，“Add Connection Point...”を選択すると，“Implement Connection Point Wizard”画面が表示されます．ここで先ほど生成したTLBファイル内で使用可能な_IMyCDInfoEventsインターフェースが候補として表示されるので，[>]ボタンを押してこれを右の“Implement connection points”欄に移動します．これでCMyCDInfo内にConnection Point Mapが作成されます．

〔図8〕でき上がったMyCDInfo.idlの内容

```
interface IMyCDInfo : IDispatch{
   [propget, id(0), helpstring("property Tracks")] HRESULT Tracks([out, retval] SHORT* pVal);
   [propput, id(0), helpstring("property Tracks")] HRESULT Tracks([in] SHORT newVal);
   [propget, id(1), helpstring("property Length")] HRESULT Length([in] SHORT Track, [out, retval] SHORT* pVal);
   [propput, id(1), helpstring("property Length")] HRESULT Length([in] SHORT Track, [in] SHORT newVal);
   [propget, id(2), helpstring("property TotalLength")] HRESULT TotalLength([out, retval] SHORT* pVal);
   [id(3), helpstring("method Read")] HRESULT Read([out,retval] SHORT* pTracks);
   [id(4), helpstring("method Redraw")] HRESULT Redraw(void);
   [id(5), helpstring("method Play")] HRESULT Play([in] SHORT Track);
};

(中略)

dispinterface _IMyCDInfoEvents
{
   properties:
   methods:
   [id(1), helpstring("method Click")] void Click([in] SHORT Track);
};
```

● イベント呼び出しの実装

前節で作成したイベントをコントロール内で実際に呼び出すには，CProxy_IMyCDInfoEvents<T>アイコンの下で表示される`Fire_Cick`関数を呼び出します．ここではコントロール内の任意のポイントでマウスを左クリックしたときに`Fire_Click`を呼び出すように実装してみます．

コントロール内でのWindowsメッセージの処理は`CMyCDInfo`内にメッセージマップを利用して実装することができます．Class View画面で`CMyCDInfo`をクリックすると，下のPropertiesペインに`CMyCDInfo`のプロパティが表示されます．ここでMessagesアイコンをクリックするとよく使うウインドウメッセージが以下に表示されます．`WM_LBUTTONDOWN`を探し，右の枠をクリックすると<Add> OnLButtonDownと表示されるので，これを選択します．

するとコードペインに`CMyCDInfo::OnLButtonDown`の処理ルーチンが表示されます．本来はメッセージのパラメータでクリックした位置を判断するのでしょうが，今回はそのまま，

```
Fire_Click(1);
```

と入力してみましょう．

以上でActiveXコントロールの作成は終わりです．“Build”メニューから“Build Solution”を選択し，MyCDInfoコントロールをビルドします．Outputウインドウに，

```
Build: 1 succeed, 0 failed, 0 skipped
```

と表示されれば成功です．この状態でMyCDInfoコントロールはシステムに登録されています．図8にでき上がった`MyCDInfo.idl`の内容の一部を表示します．

● テスト用HTMLファイルの作成

ATLウイザードでプロジェクトを作成すると，同時にテスト用HTMLファイルの雛型が作成されます．Solution Explorerペインで“`MyCDInfo.htm`”をダブルクリックするとエディタ領域にコントロールの領域を含むデザイン画面が表示されます．HTML画面を選択してHTMLコードを表示させ，OBJECTタグの下にコントロールからイベントを受け取るための以下のJScriptコードを記述して保存します[注20]．

```
<script language="JavaScript">
function MyCDInfo::Click(Track)
{
    window.alert("Clicked!");
}
</script>
```

これを実行するにはエクスプローラなどから`MyCDInfo.htm`ファイルを開きます．スクリプトエラーがなければ，(ActiveXコントロールを実行するかというダイアログが表示された後[注21])コントロールが表示されます．MyCDInfoコントロールの領域を左クリックするとメッセージボックスが表示されることと思います[注22]．

もしIEのインターネットオプションの詳細設定タブで「スクリプトのデバッグを使用しない」のチェックが入っていなければ，スクリプトエラーが起きたときにVisual Studio .NET IDEのスクリプトデバッグ機能を利用することができます．

今回作成したActiveXコントロールは，同じくVisual Studio .NETを使用してVB .NETやVisual C#で使用することができます．C#の場合には新規Windows Applicationを作成したあと，Toolboxを表示し，ToolBox領域を右クリックして表示される“Add/Remove Items...”を選択すると表示される，“Customize Toolbox”画面の“COM Components”タブ内の

注20：ちなみに，もとのCDInfo用HTMLファイルはVBScriptで記述されている．
注21：インターフェース`IObjectSafetyImpl`をクラスに継承させることによって，ダイアログを表示させないようにすることができる．
注22：いったんActiveXを実行開始すると，その実行プロセスを終了するまでその実体のDLLを上書きすることはできない．開発中に上書きできないときはすべてのIEのインスタンスを終了するか，システムを再起動する．

“ MyCDInfo Class”をチェックすることによって Toolbox に登録でき，その後，WinForms にドロップすることができます．これによって C# のフォーム内で変数 axMyCDInfo1 にアクセスすることにより，プロパティやメソッドにアクセスできます．また自動的に C# のイベント処理ルーチン axMyCDInfo1_ClickEvent が用意されます．

● iWin から ActiveX を使用する

今回のような UI をもつ ActiveX コントロールを使用する場合，普通は iWinBrowser で使用する HTML ページ内に ActiveX コントロールを配置することになります．このときはサンプル HTML と同じ OBJECT タグを作成すればよいでしょう．

UI をもたない ActiveX コンポーネントを使用する場合，iWin では二つの方法があります．

- グローバルスクリプトで生成
- ローカルスクリプトで生成

たとえば，システム中で一つしか使用しない（グローバル）オブジェクトの場合，グローバルスクリプトのスタートアップ内で以下のように生成する．

```
var MyObj = IWinSystem.CreateObject(
           "MyCDInfo.MyCDInfo.1", "MyCDInfo");
```

このようにして作成されたオブジェクトは，MyObj 変数でシステム内のどこからでもアクセスできます．ここで CreateObject の第 1 引き数は OS に登録された ActiveX の ProgID[注23]です．イベントを使用したい場合は，CreateObject の第 2 引き数を利用してイベントルーチンのプレフィックスとし，

```
function MyCDInfo_Click(track)
```

のようなグローバルスクリプト内の関数でキャッチすることができます．このように作成されたグローバルオブジェクトは iWinSystem と同一のスレッドで動作します．

ローカルスクリプトの場合は IWinBrowser.CreateObject を使用します．

● 開発機以外で動かすには

今回ビルドした ActiveX コントロールを開発機以外の PC で動かすには，Visual Studio .NET 2003のランタイムライブラリが必要になります．具体的には atl71.dll と msvcr71.dll（リリースビルドの場合）です．この二つのファイルとコントロール本体 MyCDInfo.DLL，テスト用 HTML ファイル MyCDInfo.htm の四つをコピーし，さらにコピー先で regsvr32.exe ユーティリティを利用してコンポーネントを登録することにより，開発機以外の PC で ActiveX コントロールの動作を確認できます．

じつは Visual Studio .NET 2003（Professional 以上）では Windows Installer 形式のインストーラを作成するプロジェクトがサポートされており，これを利用するとインストール時に同時にコンポーネント登録を行うようなインストーラを作成できます．

注 23：ProgID はレジストリに格納されるコンポーネントの登録名．ATL Control Wizard や .RGS ファイルに出てくる．



まとめ

iWin ソリューションはなるべく HTML とスクリプトでユーザーインターフェースを構築するように考えられたソリューションです．これは（いわゆる C++ プログラマだけではなく）Web デザイナやスクリプティングのプロフェッショナルにも組み込み機器のデザインができる可能性を開いてくれます．また，どうしても C++ でのプログラミングが不可欠な部分には ActiveX コンポーネントを作成することで代替することができ，その可能性に限りはありません．

また，今回の記事で説明したように ActiveX の作成は決して難しいものではありません．Visual Studio をはじめとするツールを利用することで，ATL の難しい部分をすべて理解しなくても実際に動作する ActiveX が作成できるのです．ぜひ iWin を利用して素晴らしい組み込み機器のデザインを実現してください．

なかやま・ひろゆき　ピースクウエア（株）

初級ドライバ開発者のための

Windowsデバイスドライバ開発テクニック

第5回 ドライバ/アプリケーションでのタイマの使い方

丸山 治雄

5.1 タイマを使う場面

● タイムアウト処理などで必要

タイマを使用するような複雑なドライバを開発する例はそれほど多くはありませんが，ドライバの動作の時間を空けたいとき，あるいはポーリングのように定期的にデータの有無を確認したいときには，タイマを使用する必要があります．アプリケーションでは，時間を空けるときはSleep()のように単純に動作を停止させたり，マルチメディアタイマ関数を使用して簡単に実現できますが，ドライバの場合は簡単にはいきません．今回はドライバで手軽にタイマを使用できるよう，簡単なサンプルプログラムをもとに解説します．

● タイマの種類

タイマには，ごく短時間だけSleep()と同様の動作をする関数，定期的にタイマ割り込みが発生するインターバルタイマ，待機関数のパラメータでタイムアウトを指定する同期関数の3種類があります．

さらに，インターバルタイマには，通知タイマと同期タイマがあります．通知タイマは，待機中のすべてのスレッドが解放され，タイマがKeSetTimer()またはKeSetTimeEx()によりタイマシグナルがリセットされるまでシグナル状態を保持します．同期タイマは，同期している一つのスレッドのみ解放されて，タイマはスレッドの解放と同時にシグナル状態をリセットします．同期タイマは，KeInitializeTimerEx()のみで，TIMER_TYPEパラメータに，SynchronizationTimerを与えることで初期化できます．通知タイマは，KeInitializeTime()か，KeInitializeTimerEx()のTIMER_TYPEパラメータに，NotificationTimerを与えることで初期化できます．

5.2 実行の遅延

ドライバ内で，ごく短時間の遅延を行うのであれば，KeStallExecutionProcessor()関数を使用することができます．ただし，この関数を実行している間はプロセッサが占有されるため，ほかのプロセスの実行に影響を与えます．そのため，マイクロソフトは，この関数による遅延の上限を50μsにするよう推奨しています．したがって1ms遅延させたいときにはこの関数を使用できないことになります．

5.3 簡単なタイマ

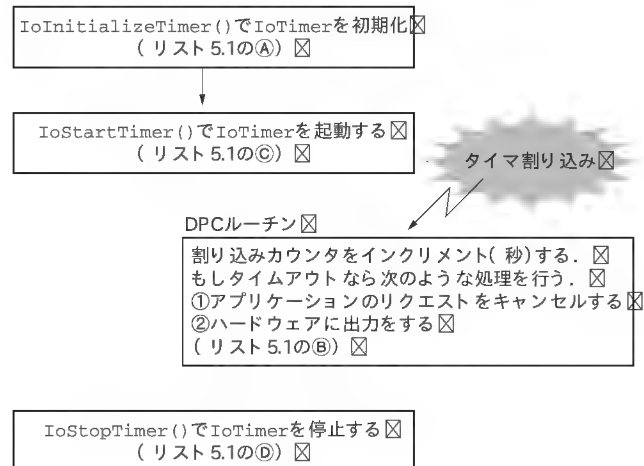
● 秒単位のタイマ

インターバルタイマで，たとえば一定時間経過しても期待したイベントが発生しない，または完了しないようなことが想定される場合で，監視の間隔が秒単位でよいときには，IoTimerルーチンを使用することができます．これは，タイマを起動すると1秒に1回の割合で，指定のDPCルーチンにタイマ割り込みが発生します．DPCルーチンではカウンタをインクリメントします．このカウンタの値が，タイマを起動してからの経過秒数を示すことになります．

● ドライバ(DPCルーチン)の中でタイマを監視する方法

使用方法としては，二つのケースが考えられます．一つはドライバ(DPCルーチン)の中でタイマを監視して，タイムアウト時間になってもイベントが発生しない，または終了していな

〔図5.1〕ドライバ(DPCルーチン)の中でタイマを監視する場合(リスト5.1)の動作



〔リスト 5.1〕ドライバ（DPC ルーチン）の中でタイマを監視する方法

```
// デバイス拡張定義の中のタイマ関連宣言
typedef struct _DEVICE_EXTENSION {
    LONG timeCount;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;


// 簡易タイマの登録
// DriverEndtry()の中で行う
    pExtension->timeCount = 0; ←Ⓐ
    IoInitializeTimer( DriverObject,
                       IntervalTimer1Sec,
                       pExtension
                       );

// 1秒単位のタイマ割り込みを受け付ける DPC ルーチン                    ┐
VOID IntervalTimer1Sec( IN PDEVICE_OBJECT DeviceObject,                │
                        IN PVOID pContext                              │
                        )                                              │
{                                                                      │
    PDEVICE_EXTENSION  pExtension = DeviceObject->Extension;           │
                                                                       │
    pExtension->timeCount++;// 1秒単位の加算                            │
    if ( pExtension->timeCount > TimeOut ) // TimeOut は任意           ├Ⓑ
    {                                                                  │
//        タイムアウトが発生                                            │
//        処理としてはアプリケーションからのリクエストをキャンセルする.  │
//        あるいはハードウェアに対して出力を行う.                        │
                                                                       │
        pExtension->timeCount = 0;                                     │
    }                                                                  │
}                                                                      ┘

NTSTATUS
KIT1050CreateClose(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PDEVICE_EXTENSION
                    pExtension  = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack;

    irpStack = IoGetCurrentIrpStackLocation(Irp);

    switch (irpStack->MajorFunction)
    {
    // タイマ起動
    case IRP_MJ_CREATE:                    ←Ⓒ
        IoStartTimer(DeviceObject);
        break;
    // タイマ停止
    case IRP_MJ_CLOSE:                     ←Ⓓ
        IoStopTimer(DeviceObject);
        break;
    }
}
```

〔リスト 5.2〕タイマの管理をアプリケーションに任せる方法

```
// デバイス拡張定義の中のタイマ関連宣言
typedef struct _DEVICE_EXTENSION {
    LONG timeCount;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;


// 簡易タイマの登録
// DriverEndtry()の中で行う
    pExtension->timeCount = 0; ←Ⓐ
    IoInitializeTimer( DriverObject,
                       IntervalTimer1Sec,
                       pExtension
                       );

//                                                                     ┐
// 1秒単位のタイマ割り込みを受け付ける DPC ルーチン                    │
VOID IntervalTimer1Sec( IN PDEVICE_OBJECT DeviceObject,                │
                         IN PVOID pContext                             │
                        )                                              ├Ⓑ
{                                                                      │
    PDEVICE_EXTENSION  pExtension = DeviceObject->Extension;           │
                                                                       │
    pExtension->timeCount++;// 1秒単位の加算                            │
}                                                                      ┘


// KIT1050 のユーザ IOCTRL ルーチン
NTSTATUS KIT1050DeviceControl(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp
    )
{
    PIO_STACK_LOCATION irpStack =
IoGetCurrentIrpStackLocation(Irp);
    PDriverRequest Request;
    Driver_Open
            *DRVHdl= (PVOID)Irp->AssociatedIrp.SystemBuffer;
    PDEVICE_EXTENSION
                pExtension = DeviceObject->DeviceExtension;
    NTSTATUS  status = STATUS_SUCCESS;// Driver status;

    ULONG     Handle = (ULONG)-1;
    ULONG     IoControlCode;
    ULONG     rc = (ULONG)-1L;         // Function status;

    LONG      OutputBufferLength;
    LONG      InputBufferLength;


    Request = (PDriverRequest)Irp->UserBuffer;
    IoControlCode =
        irpStack->Parameters.DeviceIoControl.IoControlCode;

    OutputBufferLength =
        irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    InputBufferLength =
        irpStack->Parameters.DeviceIoControl.InputBufferLength;
    Irp->IoStatus.Information = 0;

    switch( IoControlCode )
    {
    // タイマ起動
    case IOCTL_TIMER_START:
        IoStartTimer(DeviceObject); ←Ⓒ
        break;
    // タイマ停止
    case IOCTL_TIMER_STOP:
        IoStopTimer(DeviceObject); ←Ⓓ
        break;
    // タイマ値読み出し
    case IOCTL_TIMER_GETCOUNT:                                             ┐
        {                                                                  │
            PULONG                                                         │
            APmem = (PVOID)Irp->AssociatedIrp.SystemBuffer;                ├Ⓔ
            *APmem = pExtension->timeCount;                                │
            Irp->IoStatus.Information = sizeof(LONG);                      │
        }                                                                  │
        break;                                                             ┘
    }

    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (Irp, IO_NO_INCREMENT);
    return( status );
}
```

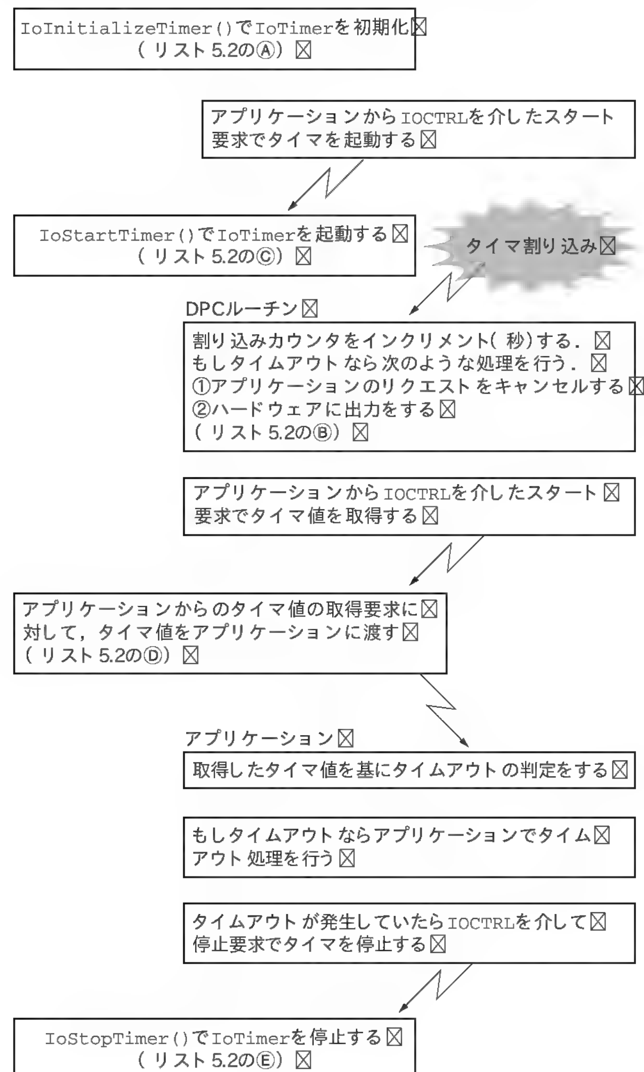
いときは，ハードウェアに対して，何らかのアクションを発生する，あるいはアプリケーションからのリクエストを強制的に終了させる方法があります(リスト5.1，図5.1)．

IoStartTimer()は，DriverEntry()の中で行っても良いのですが，一般的には，IRP_MJ_CREATE(I/Oディスパッチルーチン)で行います．また，IoStopTimer()は，DriverEntry()でタイマを起動したときは，ドライバアンロードの中で行い，IRP_MJ_CREATE(I/Oディスパッチルーチン)で起動したときは，IRP_MJ_CLOSE(I/Oディスパッチルーチン)で行います．

● タイマの管理をアプリケーションに任せる方法

もう一つの方法は，タイマの管理をアプリケーションに任せる方法です(リスト5.2，図5.2)．これは，IOCTRL(ユーザーディスパッチルーチン)を使用して，アプリケーションがドライバのタイマ管理を行うものです．ただし，この方法の欠点として，タイムアウトが発生したか否かの問い合わせを，アプリケーションがドライバに対して頻繁に行うため，動作効率が悪くなることがあげられます．また，タイムアウトが発生したときに，タイムアウトの処理をドライバで行うか，アプリケーションで行うかの切り分けが難しいという欠点があります．ただし，それほど複雑ではないので，たとえばDPCルーチンの中で，アプリケーションのリクエストを強制終了するような処理は必要ありません．ただし，IoTimerが動作中にアプリケーションが異常終了したときは，タイマを停止する処理が必要になります．この場合はIRP_MJ_CLOSEディスパッチの中に処理を入れることを勧めます．ドライバがシンプルになるので場合によっては，この方法を使用することも検討の価値があると思います．

〔図5.2〕タイマの管理をアプリケーションに任せた場合(リスト5.2)の動作



5.4 インターバルタイマ

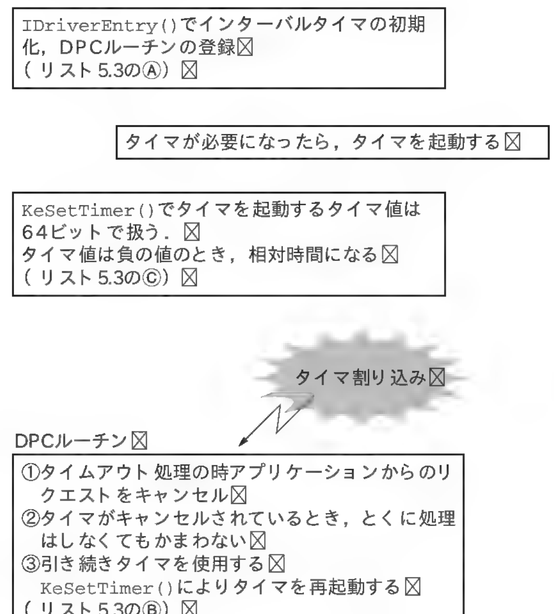
● ms単位のタイマ

1秒単位では実用的でなく，ms単位でインターバルタイマを制御したいときには，ここで説明するように独自にインターバルタイマを管理するプログラムを作成する必要があります(リスト5.3，図5.3)．まず，DriverEntry()内でインターバルタイマの初期化とDPCルーチンの登録を行います．

タイマの削除はUnload()ルーチンの中で行います．削除するときは，念のためKeCancelTimer()でタイマを停止して，タイマ用キューからタイマオブジェクトを削除してください．もしタイマが停止しているときは，この関数は処理を行わないので，問題は発生しません．

アプリケーションからのリクエストかドライバの判断で，インターバルタイマを起動します．タイマの単位はμsですが，実

〔図5.3〕ms単位でインターバルタイマを制御したい場合(リスト5.3)の動作



〔リスト 5.3〕ms 単位でインターバルタイマを制御したい場合

```
// デバイス拡張定義の中のタイマ関連宣言
typedef struct _DEVICE_EXTENSION {
    // Serial Lock to protect the timer
    KSPIN_LOCK ControlLock;

    // Timer service
    KTIMER        RequestIntervalTimer;    // Timer
    KDPC          IntervalTimeoutDpc;      // TimerのDPC
    LARGE_INTEGER IntervalTimeToUse;       // Interval timing.
    ULONG         IntervalCount;           // Interval Time Count
    LONG          CountOnLast;             // Timer Flag

} DEVICE_EXTENSION, *PDEVICE_EXTENSION;



NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    // タイマの登録                                                    ┐
    KeInitializeSpinLock( &pExtension->ControlLock );                   │
    KeInitializeTimer( &pExtension->RequestIntervalTimer );             │
    KeInitializeDpc(                                                    │
            &pExtension->IntervalTimeoutDpc,                            ├Ⓐ
            IntervalTimeout,                                            │
            pExtension                                                  │
            );                                                          ┘
}

VOID
KIT1050Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
    PDEVICE_EXTENSION pExtension
        = DriverObject->DeviceObject->DeviceExtension;

    // タイマの強制停止
    KeCancelTimer( &pExtension->RequestIntervalTimer );
    // タイマ処理の停止
    KeCancelTimer( &pExtension->RequestIntervalTimer );
    KeRemoveQueueDpc( &pExtension->IntervalTimeoutDpc );
}


// タイマ用DPCルーチン                                                 ┐
VOID                                                                    │
IntervalTimeout(                                                        │
    IN PKDPC Dpc,                                                       ├Ⓑ
    IN PVOID DeferredContext,                                           │
    IN PVOID SystemContext1,                                            │
    IN PVOID SystemContext2                                             │
    )                                                                   │
{                                                                       │

    PDEVICE_EXTENSION pExtension = DeferredContext;
    KIRQL oldIrql;

    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(SystemContext1);
    UNREFERENCED_PARAMETER(SystemContext2);

    IoAcquireCancelSpinLock(&oldIrql);

    if ( pExtension->CountOnLast ==
                            SERIAL_COMPLETE_READ_COMPLETE )
    {
        // タイムアウトが発生した
        // タイムアウトの処理を行う
    }
    else
    if ( pExtension->CountOnLast ==
                            SERIAL_COMPLETE_READ_CANCEL )
    {
        // タイムアウトが発生した
        // タイマがキャンセルさせているときの処理を行う
    }
    else
    {
        // タイマを再起動するとき(KeSetTimerで起動したときで,引き続き
        // タイマをしようするとき.KeSetTimerExで起動したときは不要)
        KeSetTimer(
            &pExtension->RequestIntervalTimer,
            pExtension->IntervalTimeToUse,
            &pExtension->IntervalTimeoutDpc
            );
        pExtension->IntervalCount++;
    }                                                                   ┘

    IoReleaseCancelSpinLock( oldIrql );

}


// タイマが必要になったとき,インターバルタイマを起動する               ┐
// 負の値のときは,相対時間                                             │
    pExtension->IntervalTimeToUse.QuadPart = -20000;// 2ms秒            │
    pExtension->IntervalCount = 0;                                      │
    KeSetTimer(                                                         │
        &pExtension->RequestIntervalTimer,                              ├Ⓒ
        pExtension->IntervalTimeToUse,                                  │
        &pExtension->IntervalTimeoutDpc                                 │
        );                                                              │
                                                                        │
    // 1回の割り込みのみでタイムアウトさせるとき                        │
    pExtension->CountOnLast = SERIAL_COMPLETE_READ_COMPLETE;            ┘
```

際には100μs単位で値を指定します．値はLARGE_INTEGERを使用します．リスト5.3の例では，2ms経過したら割り込みを発生させます．DPCルーチンでの動作は1回の割り込みで終了するように，CountOnLastに指定します．ここで使用しているフラグは独自に定義しています．

DPCルーチンでは，タイマ割り込みが発生したときにタイムアウトさせるか，引き続きタイマを起動するか，またはタイマがキャンセルされているかにより，その処理を分岐します．KeSetTime()でタイマを起動すると，1回のみのタイマ割り込みなので，引き続きインターバルタイマとして使用するときは，DPCルーチンの中で，タイマを再起動する必要があります．KeSetTimerEx()を使用すると，いわゆるウォッチドッグタイマとして動作するので，DPCルーチンの中では再起動する必要はありません．

● DPCルーチンの中でDPCとISRを同期させる必要があるとき

今回は使用していませんが，DPCルーチンの中でDPCとISRを同期させる必要があるとき(ハードウェアをリセットするために，IRQをキャンセルするときなど)は，KeInitializeDpc()のDeferredContextパラメータに，DeviceObjectを指定します．DPCルーチンでは，デバイスオブジェクトから次のように必要なパラメータを取得し，KeSynchronizeExecution()やIoCancelIrp()を使用することができます．

```
PDEVICE_OBJECT DeviceObject
            = (PDEVICE_OBJECT)DeferredContext;
PIRP Irp = DeviceObject->CurrentIrp;
```

5.5 スレッドのタイムアウト

スレッド内でイベントの発生を待つための同期関数でも，タイムアウトを使用することができます．これは，今まで説明したインターバルタイマを使用したものとは異なりますが，一定時間待機状態が終了しない(イベントが発生しない)ときにタイムアウトとして強制終了させることができます．関数を呼び出す際に，Timeoutパラメータに，タイムアウト値を指定しておくと，ここで指定した時間が経過しても待機状態が終了しないときは，STATUS_TIMEOUTの戻り値で関数が終了します．Timeoutに与える値は，インターバルタイマで使用する値と同じLARGE_INTEGERです．正の値で絶対時間を，負の値で相対時間を指定します．ここで注意することは，Timeoutのパラメータに NULLを与えることと，Timeout変数に0を与えることでは，動作内容が異なる点です．

パラメータにNULLを与えると，タイムアウトはなしとなり，タイムアウトによる打ち切りはありません．また，Timeout変数に0を与えると待機状態のポーリングになります．したがって，タイマを使用しないときはNULLを，タイマを使用するときは，Timeout変数に0以外の値を設定してTimeout変数のポインタを与えます．

タイムアウトが使用できる関数は，KeWaitForSingleObject()とKeWairForMultipleObjecs()の二つの関数です．

今回は使用していませんが，スレッド自体の実行を遅延させるという手段もあります．これはKeDelayExecutionThread()を使用することにより，スレッドが起動および実行される時間を遅くすることが可能です．

5.6 アプリケーションでインターバルタイマを使用する

次はドライバ内部ではなく，アプリケーションで使用する方法を説明します．インターバルタイマの分解能はシステムによって異なります．分解能を調べるときは，アプリケーションからtimeGetDevCaps()関数により，最小分解能と最大分解能を調べてください．インターバル時間は，ここで取得した最小分解能以下には設定できないので注意してください．また，マルチメディアタイマ関数を使用するときには，リンク時にWinmm.libを追加する必要があります．

筆者の経験では，1msの分解能は確保できるので，μs単位のタイマが必要な特殊な場合を除けば，十分なインターバル間隔だと思っています．

なお，参考までに説明すると，複数のアプリケーションがマルチメディアタイマ関数を使用し，かつ異なる最小インターバルを使用したときは，一番小さな値がインターバル値として使用されます．リスト5.4で関数を説明します．

Column デバイスマネージャを簡単に開く方法

デバイスマネージャでハードウェア情報を見ようとすると，デスクトップのマイコンピュータから，プロパティまたは管理などでデバイスマネージャを開く必要があります．これを煩わしいと思っている人は多いと思います(筆者もその一人)．そこでデスクトップ上のアイコンをクリックするだけでデバイスマネージャを簡単に開く方法を説明します．次の手順でデバイスマネージャをデスクトップに登録します．

(1) ¥Winnt¥System32にある，mmx.exeのショートカットをデスクトップに配置します．

(2) 次に，同じフォルダにdevmgmt.mscがあることを確認します．

(3) デバイスマネージャのプロパティを開き，「リンク先」の最後にdevmgmt.mscのフルパスを指定します．

たとえば，C:¥Winntにシステムが登録されている場合は，C:¥WINNT¥system32¥mmc.exeです．

C:¥WINNT¥system32¥devmgmt.mscが「リンク先」に設定されていることを確認します．以上で，デバイスマネージャを起動するアイコンをデスクトップに配置できます．なお，この操作は管理者特権(Administrator)が必要になります．

● StartIntervalTimer()

インターバルタイマを起動します．最小分解能はここで調べます．リスト5.4のⒶのtimeSetEvent(IntervalTime, 1, EventIntervalService, 0, TIME_PERIODIC)は，呼び出し側から与えられたIntervalTimemsを単位に，コールバック関数を呼び出します．分解能は1msを指定していますが，タイマを秒単位で動作させるときは，もっと大きな値でもかまいません．コールバックパラメータには0を指定していますが，コールバック関数にパラメータを渡すときはここに値を設定すると，コールバック関数のdwUserに渡されます．

● StopIntervalTimer()

インターバルタイマを停止します．タイマを停止したときは，timeEndPeriod(uPeriod)により，タイマ起動時に設定した分解能をクリアしてください．また，timeBeginPeriod(uPeriod)で設定した，分解能と同じ値を指定する必要があります．

● IntervalTimerStatus()

タイマの現在の値を読み出します．

● EventIntervalService()

タイマ割り込み処理をします．timeSetEvent()のIntervalTimeで指定した時間が経過すると，この関数が呼び出されます．また，コールバックデータは3番目の，dwUser

〔リスト 5.4〕アプリケーションで使えるタイマ関数

```
#include <windows.h>
#include <windowsx.h>

#include <stdio.h>
#include <string.h>
#include <direct.h>

#include "typedef.h"
#include "evnttime.h"

static   UINT      uTimerID = 0;  // Event ID
static   UINT      uPeriod = 1;
static   ULONG     RequestTime = 0;
static   ULONG     ElapsedTime = 0;
static   LONG      ReStartTimer = 0;

/*================================================================
    名称:   StartIntervalTimer
    機能:   プロセスのタイマを起動する
    書式:   LONG StartIntervalTimer( ULONG IntervalTime );
    パラメータ
              ULONG IntervalTime  <in>  インターバルタイム (mmSec)

    返値:     0 = 正常終了
             -1 = タイマの起動に失敗
             -2 = タイマ動作中
    備考:   この関数を使用するときは，Link 時に Winmm.Lib を使用してください
================================================================*/
LONG WINAPI StartIntervalTimer( ULONG IntervalTime )
{


    if ( uTimerID != 0 )
         return( -2 );

    if ( timeBeginPeriod( uPeriod ) != 0 )
    {
         // Can't create timer!
         return( -1 );
    }

    ElapsedTime = 0;
    RequestTime = IntervalTime;

    if (( uTimerID = timeSetEvent( IntervalTime, 1,            ┐
                                   EventIntervalService,       ├Ⓐ
                                   0, TIME_PERIODIC )) == 0 )  ┘
    {
         timeEndPeriod( uPeriod );
         return( -1 );
    }

    return( 0 );

}

/*================================================================
    名称:   StopIntervalTimer
    機能:   プロセスのタイマを停止する
    書式:   VOID StopIntervalTimer( VOID );
    パラメータ
            なし
    返値:   なし
    備考:
================================================================*/
VOID WINAPI StopIntervalTimer( VOID )
{


    if( uTimerID )
    {
         timeKillEvent( uTimerID );
         timeEndPeriod( uPeriod );
    }
    uTimerID = 0;
    ElapsedTime = 0;

}

/*============================================================
    名称:   IntervalTimerStatus
    機能:   現在のタイマ値を取得する
    書式:   ULONG IntervalTimerStatus( VOID );
    パラメータ
            なし
    返値:   現在のタイマ値
    備考:
============================================================*/
ULONG WINAPI IntervalTimerStatus( VOID )
{


    return( ElapsedTime );

}

/*============================================================
    名称:   EventIntervalService
    機能:   タイマ割り込み処理をする
    書式:   VOID CALLBACK EventIntervalService( UINT  uTimerID,
                                               UINT   uMsg,
                                               DWORD  dwUser,
                                               DWORD  dw1,
                                               DWORD  dw2 )
    パラメータ
            UINT     uTimerID
            UINT     uMsg
            DWORD    dwUser
            DWORD    dw1
            DWORD    dw2
    返値:   なし
    処理:
    備考:
============================================================*/
VOID CALLBACK EventIntervalService(
                                  UINT   uTimerID,
                                  UINT   uMsg,
                                  DWORD  dwUser,
                                  DWORD  dw1,
                                  DWORD  dw2 )
{

    // 経過時間を加算
    ElapsedTime += RequestTime;


}
```

に渡されます．サンプルでは，カウンタを加算しているだけなので，ほかの処理を追加することは問題がありませんが，処理は IntervalTime の時間以内で終了させる必要があります．

● その他の関数

ドライバでは通常使用しませんが，アプリケーションで使用すると同じように，時刻を使用することができます．

▶ KeQuerySystemTime( )

1901 年 1 月 1 日を基準にしたチック数の時間を LARGE_INTEGER で取得します．

▶ RtlTimeTOTimeFields( )

KeQuerySystemTime() で取得した時間を年月日 24 時間での時分秒に変換します(この逆を行うこともできる)．

タイマ関連のドライバ関数はこれ以外にもありますが，ドライバが通常使用するもののみ解説しました．

まるやま・はるお　ドライバ屋

フリーソフトウェア徹底活用講座

第14回 GCC2.95から追加変更のあったオプションの補足と検証（その3）

岸 哲夫

前回に引き続きGCC2.95から追加変更のあったオプションの補足と検証を行う．今回は，「コード生成規約に対するオプション」の続きを扱う．（筆者）

● -fleading-underscore

このオプション，およびその否定のオプションである-fno-leading-underscoreは，オブジェクトファイルの中でCのシンボルが表現される方法を強制的に変更します．古いアセンブラコードとのリンクをサポートします．coff環境やi386用のクロスコンパイルでも使用します．

ターゲットすべてに有効であるわけではないので，理解して使わないと混乱を招きます．

Windowsの開発言語であるVisual C++は，アンダースコア付きシンボル名を生成します．よって，Cygwin環境で使用する場合には，Visual C++で作成されたDLLの呼び出しの際に注意しなくてはなりません．

ソースと生成されたコードをリスト1，リスト2に示します．

このようにシンボル名の先頭に_（アンダースコア）が付いています．リスト3に，オプションなしで生成されたソースを示します．

● -fno-common

初期化済みでないグローバル変数をオブジェクトファイル中のbssセクションに割り当てます．

コードと生成されたソースをリスト4～リスト6に示します．

生成されたアセンブラ上で疑似命令である.bssでセクションの指定がされています（リスト6）．

こちらのアセンブラソース上では共有領域をメモリに確保する疑似命令である.commが使われています．

また，二つの異なるコンパイル単位の中でexternを使わずに同一のグローバル変数が宣言されている場合，リンク時エラーを発生させることができます．

● -fno-gnu-linker

GNUのリンカを使用しないときにこのオプションを指定します．注意すべきなのは，このオプションを指定すると，C++のコンストラクタやデコンストラクタをGNUのリンカで使われる形で出力しない点です．

● -fnon-call-exceptions

このオプションに関してはC++を説明する際に扱います．throw exceptionsをトラップするために使います．

● -funwind-tables

C++で作成されたモジュールとリンクし，なおかつ例外処理を行っている場合，このオプションを指定することが必要にな

〔リスト1〕Cのシンボルが表現される方法を強制的に変更する例（test195.c）

```
void    test(void);
int     main(void)
{
        test();
        return 0;
}
```

```
void    test(void)
{
    char a;
    a='a';
}
```

〔リスト2〕オプションを付けて生成されたアセンブラソース（test195a.s）

```
    .file    "test195.c"
    .text
.globl _main
    .type    _main, @function
_main:
    pushl    %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call _test
    movl $0, %eax
    leave
```

```
    ret
    .size    _main, .-_main
.globl _test
    .type    _test, @function
_test:
    pushl    %ebp
    movl %esp, %ebp
    subl $4, %esp
    movb $97, -1(%ebp)
    leave
    ret
    .size    _test, .-_test
    .ident   "GCC: (GNU) 3.3"
```

〔リスト3〕オプションなしで生成されたアセンブラソース（test195b.s）

```
    .file    "test195.c"
    .text
.globl main
    .type    main, @function
main:
    pushl    %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test
    movl $0, %eax
    leave
```

```
    ret
    .size    main, .-main
.globl test
    .type    test, @function
test:
    pushl    %ebp
    movl %esp, %ebp
    subl $4, %esp
    movb $97, -1(%ebp)
    leave
    ret
    .size    test, .-test
    .ident   "GCC: (GNU) 3.3"
```

〔リスト 4〕初期化済みでないグローバル変数をオブジェクトファイル中の bss セクションに割り当てる例（test196.c）

```
#include <stdio.h>
void     test1();
int      ix;
int main(int argc, char* argv[])
{
    test1();
    return    0;
}
void     test1()
{
    printf("test1¥n);
}
```

〔リスト 5〕オプションを付けて生成されたアセンブラソース（test196a.s）

```
    .file     "test196.c"
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test1
    movl $0, %eax
    leave
    ret
    .size     main, .-main
    .section  .rodata
.LC0:
    .string   "test1ta
    .text
.globl test1
    .type     test1, @function
test1:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $.LC0, (%esp)
    call printf
    leave
    ret
    .size     test1, .-test1
.globl ix
    .bss
    .align 4
    .type     ix, @object
    .size     ix, 4
ix:
    .zero     4
    .ident    "GCC: (GNU) 3.3"
```

〔リスト 6〕オプションなしで生成されたアセンブラソース（test196b.s）

```
    .file     "test196.c"
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test1
    movl $0, %eax
    leave
    ret
    .size     main, .-main
    .section  .rodata
.LC0:
    .string   "test1ta
    .text
.globl test1
    .type     test1, @function
test1:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $.LC0, (%esp)
    call printf
    leave
    ret
    .size     test1, .-test1
    .comm     ix,4,4
    .ident    "GCC: (GNU) 3.3"
```

〔リスト 7〕スタックの巻き戻しをさせる例（test197.c）

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int test1();
int test2();
int main(int argc, char* argv[])
{
    printf("%d¥n",test1());
    printf("%d¥n",test2());
    return    0;
}
int test1()
{
    return    100;
}
int test2()
{
    return    200;
}
```

〔リスト 8〕オプションを付けて生成されたアセンブラソース（test197a.s）

```
    .file     "test197.c"
    .section  .rodata
.LC0:
    .string   "%dodata
    .text
.globl main
    .type     main, @function
main:
.LFB3:
    pushl     %ebp
.LCFI0:
    movl %esp, %ebp
.LCFI1:
    subl $8, %esp
.LCFI2:
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test1
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    call test2
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
.LFE3:
    .size     main, .-main
.globl test1
    .type     test1, @function
test1:
.LFB5:
    pushl     %ebp
.LCFI3:
    movl %esp, %ebp
.LCFI4:
    movl $100, %eax
    popl %ebp
    ret
.LFE5:
    .size     test1, .-test1
.globl test2
    .type     test2, @function
test2:
.LFB7:
    pushl     %ebp
.LCFI5:
    movl %esp, %ebp
.LCFI6:
    movl $200, %eax
    popl %ebp
    ret
.LFE7:
    .size     test2, .-test2
    .section
.eh_frame,"a",@progbits
.Lframe1:
    .long     .LECIE1-.LSCIE1
.LSCIE1:
    .long     0x0
    .byte     0x1
    .string   ""
    .uleb128 0x1
    .sleb128 -4
    .byte     0x8
    .byte     0xc
    .uleb128 0x4
    .uleb128 0x4
    .byte     0x88
    .uleb128 0x1
    .align 4
.LECIE1:
.LSFDE1:
    .long     .LEFDE1-.LASFDE1
.LASFDE1:
    .long     .LASFDE1-.Lframe1
    .long     .LFB3
    .long     .LFE3-.LFB3
    .byte     0x4
    .long     .LCFI0-.LFB3
    .byte     0xe
    .uleb128 0x8
    .byte     0x85
    .uleb128 0x2
    .byte     0x4
    .long     .LCFI1-.LCFI0
    .byte     0xd
    .uleb128 0x5
    .align 4
.LEFDE1:
.LSFDE3:
    .long     .LEFDE3-.LASFDE3
.LASFDE3:
    .long     .LASFDE3-.Lframe1
    .long     .LFB5
    .long     .LFE5-.LFB5
    .byte     0x4
    .long     .LCFI3-.LFB5
    .byte     0xe
    .uleb128 0x8
    .byte     0x85
    .uleb128 0x2
    .byte     0x4
    .long     .LCFI4-.LCFI3
    .byte     0xd
    .uleb128 0x5
    .align 4
.LEFDE3:
.LSFDE5:
    .long     .LEFDE5-.LASFDE5
.LASFDE5:
    .long     .LASFDE5-.Lframe1
    .long     .LFB7
    .long     .LFE7-.LFB7
    .byte     0x4
    .long     .LCFI5-.LFB7
    .byte     0xe
    .uleb128 0x8
    .byte     0x85
    .uleb128 0x2
    .byte     0x4
    .long     .LCFI6-.LCFI5
    .byte     0xd
    .uleb128 0x5
    .align 4
.LEFDE5:
    .ident    "GCC: (GNU) 3.3"
```

ります．

使用している関数に対してフレーム解放のための情報が生成されます．

-fexceptionsとの違いは，ライブラリ関数に対してはその情報が生成されないことです．

ただし，CやC++のすべてのモジュールがGCCで作成された場合，それを意識しなくても良いはずです．そして，まったく別の環境で作成されたオブジェクトファイルとリンクさせる場合にも，その言語環境で同じような機能を用意しているかもしれません．

コードと生成されたソースをリスト7～リスト9に示します．このようにフレーム解放のための情報が生成されます．

● -fasynchronous-unwind-tables

フレーム解放のための情報をdwarf2フォーマットで生成することができます．もちろん，ターゲット側がdwarf2フォーマットに対応していることが前提となります．

そして，非同期イベントが発生したときにフレームの巻き戻しを行うことが可能になります．

基本的に，-funwind-tablesと同様の機能です．

● -finstrument-functions

このオプションを付けると，関数の出口と入口でトレース用の関数を呼び出す命令を付加します．関数の入口では，

```
void __cyg_profile_func_enter (
            void *this_fn,void *call_site);
```

以上を，関数の出口では，

```
void __cyg_profile_func_exit (
            void *this_fn,void *call_site0);
```

以上を，それぞれ呼び出します．

この関数をトレースに利用することができます．しかし，単純なトレースならばGDBを使ったほうが早いと思います．

コードをリスト10に示します．

生成されたソースを後で示しますが，付加された関数，

```
void __cyg_profile_func_enter(
            void *this_fn,void *call_site)
```

以上が関数の入り口で呼び出されるので，次のように情報を取得できます．

当該関数のアドレスを標準出力に，

```
printf("%p¥n",this_fn);
```

以上を，当該関数から戻った箇所を標準出力に，

```
printf("%p¥n",call_site);
```

以上を，それぞれ出力しています．

コンパイルと実行の結果を次に示します．

```
$ gcc test198.c -o test198
        -finstrument-functions  -Xlinker
            -Map -Xlinker test198.map
```

上のようにコンパイルすると，マップファイル(リスト11)が出力され，実行結果は次のようになります．

〔リスト9〕オプションなしで生成されたアセンブラソース(test197b.s)

```
    .file    "test197.c"
    .section .rodata
.LC0:
    .string  "%dodata
    .text
.globl main
    .type    main, @function
main:
    pushl    %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test1
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    call test2
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
    .size    main, .-main
.globl test1
    .type    test1, @function
test1:
    pushl    %ebp
    movl %esp, %ebp
    movl $100, %eax
    popl %ebp
    ret
    .size    test1, .-test1
.globl test2
    .type    test2, @function
test2:
    pushl    %ebp
    movl %esp, %ebp
    movl $200, %eax
    popl %ebp
    ret
    .size    test2, .-test2
    .ident   "GCC: (GNU) 3.3"
```

〔リスト10〕関数の出口と入り口に関数呼び出しを付加する例(test198.c)

```
#include <stdio.h>
int test1();
int test2();
int test21();
int main(int argc, char* argv[])
{
    printf("%d¥n",test1());
    printf("%d¥n",test2());
    return   0;
}
int test1()
{
    return   100;
}
int test2()
{
    printf("%d¥n",test21());
    return   200;
}
int test21()
{
    return   300;
}
void __cyg_profile_func_enter(void *this_fn,void *call_site)
                           __attribute__((no_instrument_function));
void __cyg_profile_func_enter(void *this_fn,void *call_site)
{
    printf("%p¥n",this_fn);
    printf("%p¥n",call_site);
}
```

```
$ ./test198
0x80483d4
0x42015574
0x8048441
0x80483fd
100
0x804847b
0x8048412
0x80484ca
0x804849a
300
```

〔リスト 11〕test198のマップファイル(test198.map)

```
Archive member included because of file (symbol)

/usr/lib/libc_nonshared.a(elf-init.oS)
                              /usr/lib/crt1.o (__libc_csu_init)

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD /usr/lib/crt1.o
LOAD /usr/lib/crti.o
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/crtbegin.o
LOAD /tmp/cc6CRRN2.o
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/libgcc.a
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/libgcc_eh.a
LOAD /usr/lib/libc.so
START GROUP
LOAD /lib/libc.so.6
LOAD /usr/lib/libc_nonshared.a
END GROUP
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/libgcc.a
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/libgcc_eh.a
LOAD /usr/local/gccbinutils/lib/gcc-lib/i686-pc-linux-gnu/3.3/crtend.o
LOAD /usr/lib/crtn.o
                0x080480f4                . = (0x8048000 + SIZEOF_HEADERS)

.interp         0x080480f4       0x13
 *(.interp)
 .interp        0x080480f4       0x13 /usr/lib/crt1.o

.note.ABI-tag   0x08048108       0x20
 .note.ABI-tag  0x08048108       0x20 /usr/lib/crt1.o

.hash           0x08048128       0x30
 *(.hash)
                                ～以下略～
```

〔リスト 12〕test198の逆アセンブラリスト(test198.txt)

```
a.out:     ファイル形式 elf32-i386

セクション .init の逆アセンブル:

080482b0 <_init>:
 80482b0:     55                        push   %ebp
 80482b1:     89 e5                     mov    %esp,%ebp
 80482b3:     83 ec 08                  sub    $0x8,%esp
 80482b6:     e8 79 00 00 00            call   8048334 <call_gmon_start>
 80482bb:     e8 e0 00 00 00            call   80483a0 <frame_dummy>
 80482c0:     e8 db 02 00 00            call   80485a0 <__do_global_ctors_aux>
 80482c5:     c9                        leave
 80482c6:     c3                        ret
セクション .plt の逆アセンブル:

080482c8 <.plt>:
 80482c8:     ff 35 ec 96 04 08         pushl  0x80496ec
 80482ce:     ff 25 f0 96 04 08         jmp    *0x80496f0
 80482d4:     00 00                     add    %al,(%eax)
 80482d6:     00 00                     add    %al,(%eax)
 80482d8:     ff 25 f4 96 04 08         jmp    *0x80496f4
 80482de:     68 00 00 00 00            push   $0x0
 80482e3:     e9 e0 ff ff ff            jmp    80482c8 <_init+0x18>
 80482e8:     ff 25 f8 96 04 08         jmp    *0x80496f8
 80482ee:     68 08 00 00 00            push   $0x8
 80482f3:     e9 d0 ff ff ff            jmp    80482c8 <_init+0x18>
 80482f8:     ff 25 fc 96 04 08         jmp    *0x80496fc
 80482fe:     68 10 00 00 00            push   $0x10
 8048303:     e9 c0 ff ff ff            jmp    80482c8 <_init+0x18>
セクション .text の逆アセンブル:

08048310 <_start>:
 8048310:     31 ed                     xor    %ebp,%ebp
 8048312:     5e                        pop    %esi
 8048313:     89 e1                     mov    %esp,%ecx
 8048315:     83 e4 f0                  and    $0xfffffff0,%esp
 8048318:     50                        push   %eax
 8048319:     54                        push   %esp
                                ～以下略～
```

```
200
$
```

また，次のようにコマンドを入力するとtest198.txt(リスト 12)に実行形式の逆アセンブラリストが出力されます．

```
$ objdump -d a.out > test198.txt
```

オブジェクトファイルのシンボルを表示したい場合は次のようにコマンドを入力します．test198.lst(リスト 13)に出力されます．

```
$ nm -a test198 > test198.lst
```

実行結果とtest198.txtを付き合わせると次のことがわかります．

0x80483d4　関数mainのアドレス
0x42015574　戻った先の次の命令
0x8048441　関数test1のアドレス
0x80483fd　戻った先の次の命令
100　main中での最初のprintfの結果
　　（関数test1から抜けた）
0x804847b　関数test2のアドレス
0x8048412　戻った先の次の命令
0x80484ca　関数test21のアドレス
0x804849a　戻った先の次の命令
300　関数test2中でのprintfの結果
　　（関数test21から抜けた）
200　関数main中での2番目のprintfの結果
　　（関数test2から抜けた）

このようにトレースに使うことができます．関数の作り方によっては実行時間の計測も可能です．リスト 10から生成されたソースをリスト 14に示します．

次のような行が付加されています．

```
call  __cyg_profile_func_enter
call  __cyg_profile_func_exit
```

なお，比較対象として，オプションなしで生成したソースをリスト 15に示します．

● -fshort-wchar

wchar_t型は通常4バイトです．しかし，Windowsではwchar_t型をshort unsigned int(2バイト)として扱わなければなりません．このオプションを指定すると，wchar_t型を2バイトとして扱います．Cygwinなどで Windows環境とリンクする場合，またはWINEなどのWindowsエミュレータを使う場合に使用します．

コードと生成されたソースをリスト 16，リスト 17に示します．また，オプションなしでコンパイルした結果をリスト 18に示します．

リスト 17からわかるように，変数dataは内部

〔リスト 13〕test198のシンボルリスト(test198.lst)

```
080480f4 r
08048108 r
08048128 r
08048158 r
080481c8 r
08048250 r
08048260 r
08048290 r
08048298 r
080482b0 t
080482c8 t
08048310 t
080485d0 t
080485ec r
080485fc r
08049600 d
0804960c d
080496d4 d
080496dc d
080496e4 d
080496e8 d
08049704 b
00000000 ?
00000000 N
00000000 N
00000000 N
00000000 N
00000000 N
00000000 N
00000000 N
00000000 a
00000000 a
00000000 a
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/config.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/config.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/config.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/config.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/config.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/abi-tag.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crti.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crti.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crti.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crtn.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crtn.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/crtn.S
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/defs.h
00000000 a /usr/src/build/231499-i386/BUILD/glibc-2.3.2-20030313/build-i386-linux/csu/defs.h
00000000 a <built-in>
00000000 a <built-in>
00000000 a <built-in>
00000000 a <built-in>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
00000000 a <command line>
```

～以下略～

〔リスト 14〕オプションを付けて生成されたアセンブラソース(test198a.s)

```
    .file     "test198.c"
.globl __cyg_profile_func_enter
.globl __cyg_profile_func_exit
    .section  .rodata
.LC0:
    .string   "%dodata
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    pushl     %ebx
    subl $20, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $main, (%esp)
    call __cyg_profile_func_enter
    call test1
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    call test2
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %ebx
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $main, (%esp)
    call __cyg_profile_func_exit
    movl %ebx, %eax
    movl -4(%ebp), %ebx
    leave
    ret
    .size     main, .-main
.globl test1
    .type     test1, @function
test1:
    pushl     %ebp
    movl %esp, %ebp
    pushl     %ebx
    subl $20, %esp
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test1, (%esp)
    call __cyg_profile_func_enter
    movl $100, %ebx
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test1, (%esp)
    call __cyg_profile_func_exit
    movl %ebx, %eax
    addl $20, %esp
    popl %ebx
    popl %ebp
    ret
    .size     test1, .-test1
.globl test2
    .type     test2, @function
test2:
    pushl     %ebp
    movl %esp, %ebp
    pushl     %ebx
    subl $20, %esp
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test2, (%esp)
    call __cyg_profile_func_enter
    call test21
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $200, %ebx
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test2, (%esp)
    call __cyg_profile_func_exit
    movl %ebx, %eax
    addl $20, %esp
    popl %ebx
    popl %ebp
    ret
    .size     test2, .-test2
.globl test21
    .type     test21, @function
test21:
    pushl     %ebp
    movl %esp, %ebp
    pushl     %ebx
    subl $20, %esp
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test21, (%esp)
    call __cyg_profile_func_enter
    movl $300, %ebx
    movl 4(%ebp), %eax
    movl %eax, 4(%esp)
    movl $test21, (%esp)
    call __cyg_profile_func_exit
    movl %ebx, %eax
    addl $20, %esp
    popl %ebx
    popl %ebp
    ret
    .size     test21, .-test21
    .section  .rodata
.LC1:
    .string   "%podata
    .text
.globl __cyg_profile_func_enter
    .type     __cyg_profile_func_enter,
                                      @function
__cyg_profile_func_enter:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    movl $.LC1, (%esp)
    call printf
    movl 12(%ebp), %eax
    movl %eax, 4(%esp)
    movl $.LC1, (%esp)
    call printf
    leave
    ret
    .size     __cyg_profile_func_enter,
                 .-__cyg_profile_func_enter
    .ident    "GCC: (GNU) 3.3"
```

［リスト 15］オプションなしで生成されたアセンブラソース（test198b.s）

```
    .file     "test198.c"
    .section  .rodata
.LC0:
    .string   "%dodata
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    call test1
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    call test2
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
    .size     main, .-main
.globl test1
    .type     test1, @function
test1:
    pushl     %ebp
    movl %esp, %ebp
    movl $100, %eax
    popl %ebp
    ret
    .size     test1, .-test1
.globl test2
    .type     test2, @function
test2:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    call test21
    movl %eax, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $200, %eax
    leave
    ret
    .size     test2, .-test2
.globl test21
    .type     test21, @function
test21:
    pushl     %ebp
    movl %esp, %ebp
    movl $300, %eax
    popl %ebp
    ret
    .size     test21, .-test21
    .section  .rodata
.LC1:
    .string   "%podata
    .text
.globl __cyg_profile_func_enter
    .type     __cyg_profile_func_enter, @function
__cyg_profile_func_enter:
    pushl     %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    movl $.LC1, (%esp)
    call printf
    movl 12(%ebp), %eax
    movl %eax, 4(%esp)
    movl $.LC1, (%esp)
    call printf
    leave
    ret
    .size     __cyg_profile_func_enter, .-__cyg_profile_func_enter
    .ident    "GCC: (GNU) 3.3"
```

［リスト 16］wchar_t型のサイズを変更する例（test199.c）

```
/*
 *wchar_t のサイズ
 */
#include <stdio.h>
int main(int argc, char* argv[])
{
    wchar_t   data =    0;
    printf("wchar_t のサイズは %dです¥n",sizeof(data));
    return    0;
}
```

で 2バイトとして扱われています．

そして，変数 data は内部で 4バイトとして扱われています．

コンパイルと実行の結果を次に示します．

```
$ gcc test199.c
$ ./a.out
wchar_t のサイズは 4 です
$ gcc test199.c  -fshort-wchar
$ ./a.out
wchar_t のサイズは 2 です
```

＊　　　＊

次号では引き続き「コード生成規約に対するオプション」の補足，および「最適化オプション」の補足を行います．

きし・てつお

［リスト 17］オプションを付けて生成されたアセンブラソース（test199a.s）

```
    .file     "test199.c"
    .section  .rodata
.LC0:
    .string   "wchar_t
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    subl $24, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    movw $0, -2(%ebp)
    movl $2, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
    .size     main, .-main
    .ident    "GCC: (GNU) 3.3"
```

［リスト 18］オプションなしで生成されたアセンブラソース（test199b.s）

```
    .file     "test199.c"
    .section  .rodata
.LC0:
    .string   "wchar_t
    .text
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl %esp, %ebp
    subl $24, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    movl $0, -4(%ebp)
    movl $4, 4(%esp)
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
    .size     main, .-main
    .ident    "GCC: (GNU) 3.3"
```

小型・軽量でSDメモリカードと互換性もある

SDIOカード開発入門

第4回 SDIOカード設計事例（前編）

八十島 広至

はじめに

今回と次回の2回にわたって，SDIOカードの設計および開発方法について具体的に解説する．“SDIOカードを設計する”とは，SDIOカードのハードウェア設計は当然として，それだけではなく，ドライバソフトウェアの設計も含まれる．ハードウェア設計には，その動作試験環境，すなわちSDIOホストと動作させるためのドライバソフトウェアおよびアプリケーションソフトウェアを準備しなければならない．

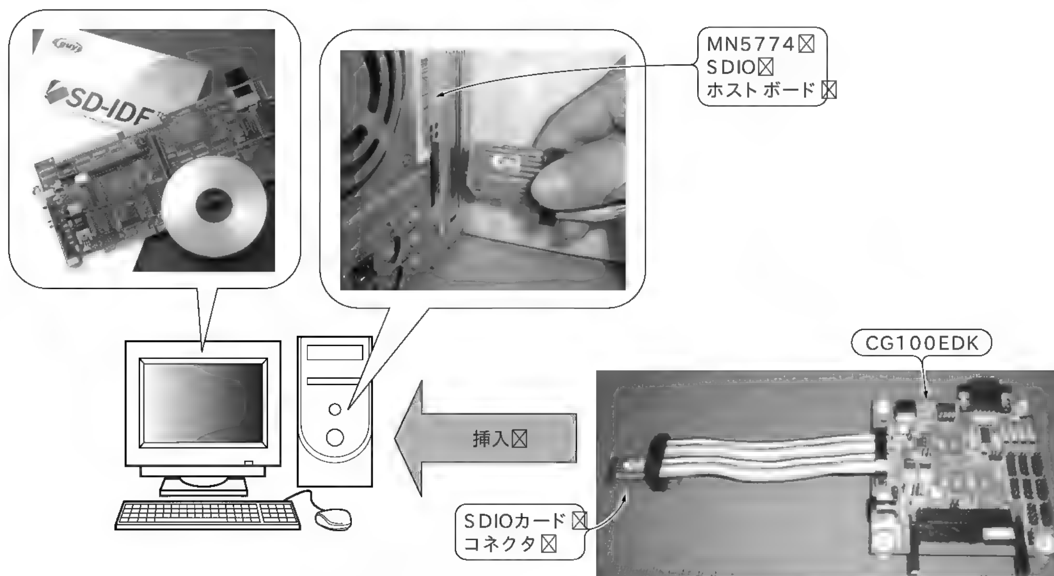
また，ドライバソフトウェアの設計には，その動作試験環境，すなわちターゲットとなるSDIOホストとアプリケーションソフトウェア，およびそのドライバソフトウェアで動かそうとしているSDIOカードを準備しなければならない．連載の第1回でも触れたように，SDIOカードを開発するには，SDIOカード側の開発環境とホスト側の開発環境が必要となる．

今回は，SDIOカードのハードウェアのプロトタイプ開発を中心に解説し，次回ではその動作テストを中心に解説するとともに，ドライバ開発についても触れる．今回の解説で，SDIOカードがホストからどのように見えるのか，また次回の解説で，どのようなシーケンス，およびコマンドでSDIOカードが制御されるのかを理解してもらえれば幸いである．

何もないところからスクラッチで開発することも可能だが，ここでは，SDIO開発環境ツールのSD-IDE（シイガイズ，図1）を使用する．現在のところ，SDIO関係の開発環境は，それほど充実していないのが実情だが，そのような中でSD-IDE/SD-EDKは，SDIOカードのプロトタイプ開発に有力なツールである．これを使用することにより，ホスト側のハードウェア環境，すなわちSDホストコントローラと，カード側のハードウェア環境の一部，すなわちSDIOインターフェース部が揃うので，あとはカード側のI/Oファンクション部（連載の第1回で「アプリケーションチップセット」と呼んだ部分）のハードウェア設計・開発およびホスト側でターゲットとなるSDIOカードを動作させるためのドライバ開発に専念すればよい．

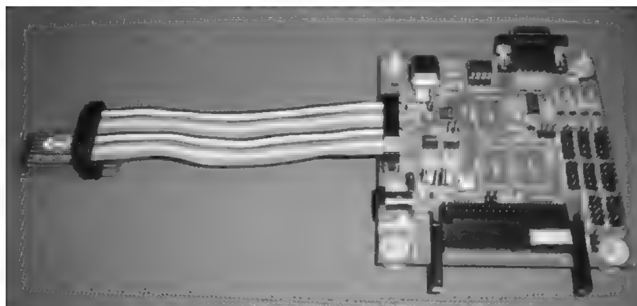
SDIOカードは非常に小さいので，試作段階からカード形状の試作品を作るようなことはせず，デバッグしやすい大きさのボードを作って機能テストをするのが一般的である．そしてそ

〔図1〕SDIO開発キット

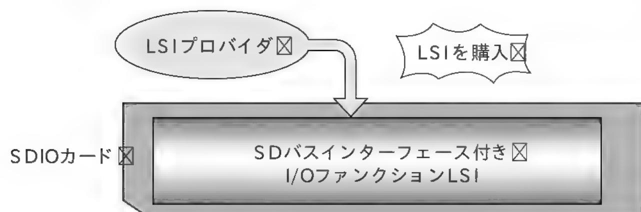


注：SD-IDEは，SD-IDEソフトウェアと開発ボードCG100EDKで構成されており，別売は不可

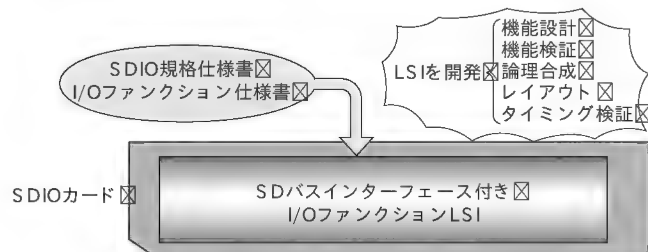
〔写真1〕開発ボードとSDIOカードコネクタプラグ



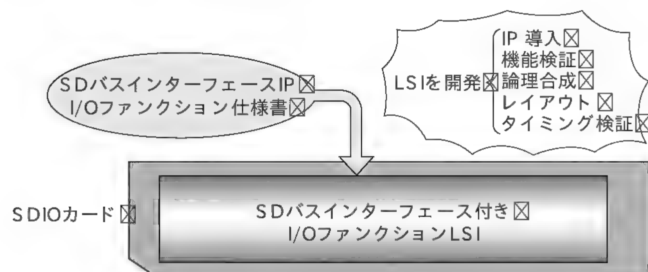
〔図2〕SDバスインターフェースをもつI/OファンクションLSIを使用する



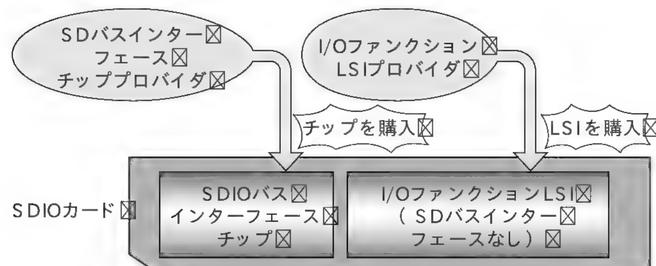
〔図3〕SDバスインターフェースをもつI/OファンクションLSIを開発する



〔図4〕SDバスインターフェース部のIPを入手してI/OファンクションLSIを開発する



〔図5〕SDバスインターフェースのコントローラチップとI/OファンクションLSIを組み合わせる



の場合には，写真1のようにボード上のSDIOインターフェースから配線コードを伸ばし，SDIOカード形状のコネクタプラグに接続する方法が扱いやすい．一般には，SDIOカード形状のコネクタプラグを入手するのは困難だが，SD-IDE/SD-EDKにはそれも含まれているので，すぐにPDAなどの既存のSDIOスロットに試作ボードを接続することができる．

なお，SD-IDEに関する詳細な情報は，次のURLを参照してほしい．

http://www.c-guys.jp/

SDIOカードの設計・開発方針

SDIOカードを設計するアプローチとしては，次の四つの方法が考えられる．

1) SDバスインターフェースをもつI/OファンクションLSIを使用する（図2）
2) SDバスインターフェースをもつI/OファンクションLSIを開発する（図3）
3) SDバスインターフェース部のIPを入手し，I/OファンクションLSIを開発する（図4）
4) SDバスインターフェースのコントローラチップとI/OファンクションLSIを組み合わせる（図5）

もっとも簡単かつ短期間でSDIOカードを設計・開発できるのは，1)の方法だが，現時点では現実的ではない．というのも，SDIOホスト側のインターフェースをもつチップは数種類あるが，SDIOカード側のインターフェースをもつチップはほとんどなく，これから開発しようとしているI/Oファンクションに合致するチップを見つけるのは，たいへん困難な状況にあるからである．

2)および3)は，SDバスインターフェース部を自分で設計する場合の設計期間も含めたコストと，SDIOバスインターフェース部のIPを入手するコストの違いはあるが，本質的には同じである．すなわち，2)と3)は，4)に比べると一般的に面積的にも性能的にも有利だが，SDバスをもつI/OファンクションLSIを新たに開発しなければならない．今後，SDバスインターフェースの利用が増えれば，これらのアプローチが主流になると思われるが，そのためにはこのようなシステムLSIが数多く開発されることが必要である．現在はその過渡期であり，目標とする性能，コスト，スケジュール（Time To Market）を十分に考慮してこのアプローチを採用すべきである．

4)の場合は，Time To Marketの観点から推奨されるアプローチであり，面積的，性能的には他の方法よりも不利になるが，SDIOカードコントローラチップとI/OファンクションLSIのインターフェースが合えば，新規にLSIを開発しなくても既存のチップの組み合わせでSDIOカードを実現できる．したがって，短期間にSDIOカードを設計開発するには，現実的かつ最良なアプローチである．SDIOカードコントローラチップ

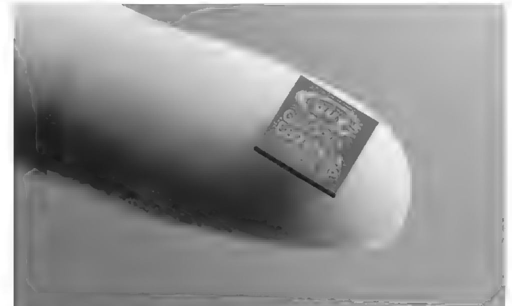
とは，SDバスとI/OファンクションLSIを接続するための，いわゆるバスブリッジのような機能をもつチップである．

SDIOカードコントローラチップCG100

SDIOカードコントローラチップとして，CG100（シイガイズ，写真2）が入手できる．CG100のブロック図を図6に示す．CG100はI/OファンクションLSIとのインターフェースとして，UART（RS-232-C），PC Card（PCMCIA），8ビットGPIOをもつ．また，カード情報などを記憶するためのEEPROMチップとのインターフェースをもつ．ノートパソコンなどでは，PC Cardスロットに何が挿入されるかわからないが，CG100の場合はPC Cardインターフェースを通して特定のI/OファンクションLSIと接続され，最終的にはSDIOカードに封入されてしまう．したがって，PC Cardインターフェースに特有のものでSDIOカードで不要な機能はサポートしていない．

ブロック転送の場合にデータは，各インターフェースで共用される2Kバイトのライトバッファ，UARTのリードデータ用512バイトのリードバッファ，PCMCIA用の2Kバイトのリードバッファ，UARTのリードデータ以外やFN0用の2Kバイトのリードバッファのいずれかを経由する．バイト転送の場合には，上記のバッファは経由せず，レジスタや各インターフェースとSDバス間で直接的にデータ転送される．また，UART，PC Card WrapperやI^{2}C EEPROMインターフェースは，それぞれDMAをもっている．

〔写真2〕SDIOカードコントローラチップCG100



〔図6〕CG100ブロック図

GPIO(7:0)
NDTR NDSR NDCD NRI SOUT SIN NCTS NRTS
命令用RAM 8Kバイト
データ用RAM 256バイト
CCLK
NCRST
CMS(1:0)
CKSEL(1:0)
MCU
GPIO
UART (FN1)
PC Card Wrapper (FN2)
NOE NPWE NIORD NIOWR NREG NWAIT NIREQ NSCHG
NCE1 NCE2 A(19:0) D(15:8) D(7:0)
SDIOREG(FN0)
TMS
SCLK SCMD SDAT0 SDAT1 SDAT2 SDAT3
SD インターフェース
リードFIFO1 512バイト
FN1：RdData
リードFIFO2 2Kバイト
FN2
リードFIFO3 2Kバイト
FN0/FN1
ライトFIFO 2Kバイト
Tx/Rx FIFO コントローラ (DMA)
I^{2}C EEPROM
SDA SCL

CG100には8051上位互換の8ビットMCUが内蔵されていて，そのファームウェアも前記のカード情報といっしょに，外付けのメモリ(EEPROMなど)に格納する．なお，ファームウェアのソースコードは公開されていない．もちろん，LSIの購入時に付いてくるファームウェアでも通常のSDIOカードの実現に支障はない．しかし，ファームウェアをそれぞれのSDIO機能に最適化することで，SDIOカードのパフォーマンスを改善することも可能になっている．

たとえば，予約領域に新規にレジスタを割り当て，UARTやPC Card上に特別なデータ転送を行うようなことも可能になる．とくにパフォーマンスが要求される場合や，複雑な制御が必要な場合などで，ファームウェアの改変が強く要望される場合は，チップベンダ(シイガイズ)に相談されたい．なお，内蔵マイコン用のデータ用メモリは256バイト，命令用メモリは8Kバイトあり，他にCG100のさまざまなレジスタへの直接アクセスが可能となっている．ファームウェアは外部メモリに格納されるが，電源投入されるとすぐに外部メモリからCG100内蔵の命令用メモリに読み込まれ，以降は内蔵の命令用メモリを使用する．ファームウェアの読み込みはハードウェアで自動的に行われるので，ファームウェアをダウンロードするためのプログラムを用意する必要はなく，そのようなプログラムを破壊するといった心配もいらない．

動作周波数は，SDバスから供給されるクロックを使うか，SDIOカード内のオシレータから供給されるクロックを使うかをダイナミックに切り替えられるが，SDバスから供給されるクロックの場合は最大25MHz，SDIOカード内のオシレータから供給されるクロックの場合は最大33MHzまで動作する．また，CG100はクロック分周器を内蔵しており，分周比をダイナミックに変更することはできないが，どちらのクロックソースを選択した場合でも，クロックソースの1/1分周，1/2分周，1/4分周のクロックをシステムクロック(内蔵マイクロコントローラ用クロック)として選択できる．

SDIOカード内のオシレータは，CG100とI/Oファンクション LSIの両方で共有するのが面積的にもコスト的にも有利だが，両チップが必要とする周波数が異なっている場合に，先に説明したクロック分周器を使用することで，両チップの動作周波数の違いを吸収したり，あるいはCG100の動作周波数を必要最低限まで落として，低消費電力化することが可能となる．

このCG100を搭載したSDIOカード開発ボードCG100-EDK(写真3)とSDIO開発ツールSD-IDEの組み合わせにより，効率的なSDIOカードプロトタイプ開発ツールキットが実現できる．UART，PC Cardなどのデバイスには精通しているが，SDIOは初めてというようなエンジニアが，SDIOカードを開発する際，この構成は最適と考えている．

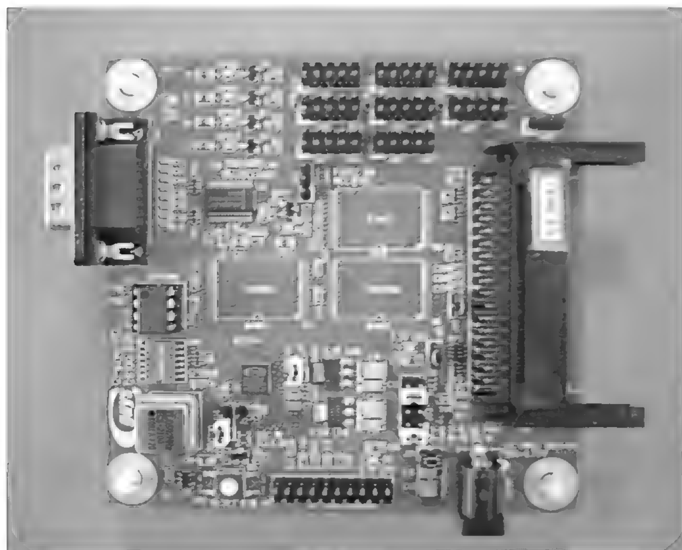
当然，CG100-EDKには，CG100がもっているすべてのインターフェースが用意されているので，これらの各種インターフェースを通して既存のI/Oファンクション LSIと接続するだけで，SDIOカードプロトタイプのハードウェア部分が完成する(図7)．なお，CG100EDKに関する詳細は，SD-IDEに附属の解説書を参照してほしい．

 CG100-EDKの概要

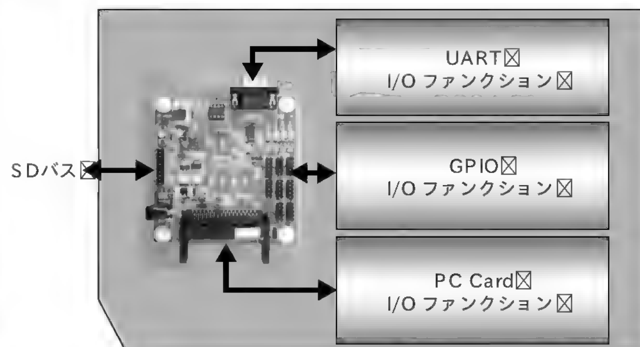
CG100はチップ単体なので，それだけ手元にあってもすぐには使えない．このチップをすぐに評価できるように用意されたボードがCG100-EDKである．

CG100-EDKのボードサイズは縦97mm×横102mmの大きさ(突起部は含まず)である．写真3の左に見えるのがUARTコネクタ，右に見えるのがPC Cardスロットである．GPIO用の特別なコネクタはなく，上に見えるヘッダピンにアサインされている．なお，CG100のすべての信号とボード上の主要な信号をヘッダピンからモニタできる．下に見えるヘッダピンがSDバスインターフェースであり，これにSDIOカード形状のコネクタプラグが接続される．

〔写真3〕SDIOカード開発ボード CG100-EDK



〔図7〕SDIOカードプロトタイプ



〔表1〕ディップスイッチとジャンパの設定

SW1	SW2	システムクロック分周比
0	0	1/1
0	1	1/2
1	0	1/4
1	1	予約

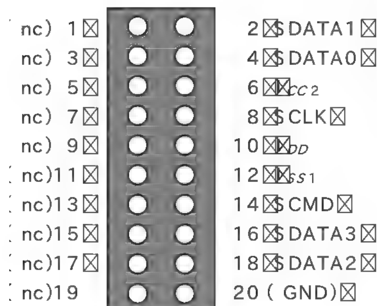
SW3	SW4	SW5	メモリ選択／モード
0	0	0	I²C EEPROM
0	0	1	NOR型フラッシュメモリ
0	1	0	NAND型フラッシュメモリ
0	1	1	予約
1	×	×	テストモード

（a）ディップスイッチの設定

JU1	外部電源入力（電流測定用）
	1：ボード上の電源レギュレータへ
	2：DC入力コネクタへ
JU3	電源選択
	1-2 & 7-8：SDインターフェースから供給
	2-3 & 9-10：外部電源から供給
JU4	システムクロック源選択
	1-2：ボード上のオシレータから供給
	2-3：SDクロックから供給
JU5	NOR型フラッシュメモリ選択
	1-2：NOR型フラッシュメモリを使用しない
	2-3：NOR型フラッシュメモリを使用する
	※I²C EEPROMを使用する場合は、1-2を接続する
JU6	PC Cardデータラインプルアップ
	1-2：プルアップする
	開放：プルアップしない
JU7	CG100 I/O電源入力（電流測定用）
	1：ボード上の3.3V電源ラインへ
	2：CG100のI/O用電源ピンへ
JU8	PC Card電源入力（電流測定用）
	1：ボード上の3.3V電源ラインへ
	2：PC Cardコネクタの電源ピンへ

（b）ジャンパの設定

〔図8〕SDインターフェースのヘッダピン配置



右下に見えるのは，外部電源用の入力コネクタだが，ジャンパの設定により外部電源を使用しないモードに設定，すなわちSDバスから供給される電源による動作も可能である．なお，外部から電源供給をする場合には，直流4.75V～6Vを入力する．また電流は1A以上の容量のものを推奨する．CG100-EDK単体でUART駆動時の消費電流は60mAくらいだが，これに接続したI/Oファンクション機能に要する電力により，外部電源なしでは安定動作しない可能性もあるので注意されたい．表1にディップスイッチおよびジャンパの設定を示す．

左上に四つのLEDが装備されているが，これらは上からGPIO[0]，GPIO[1]，GPIO[2]，GPIO[3]に接続されている．

SDインターフェースのヘッダピンの配置を図8に示す．20番ピンはGNDだが，SDインターフェースは9ピンなので，SDインターフェースには接続されない信号である．

UARTのドライバにはMAX3245が使用されていて，UARTデータ通信の最高速度は1Mbpsである．デフォルトのUARTデータ通信速度は115.2kbpsに設定されている．もちろんレジスタの設定で変更できるが，さらに，CG100-EDK上のオシレータを交換することで，さまざまな通信速度を実現できる．出荷時には11.0592MHzのオシレータが装着されているが，これはソケット装着になっているので，容易に交換が可能である．UARTの通信速度に限らず，システムクロック，すなわちCG100内蔵のマイクロコントローラの動作周波数やPC Cardインターフェースの基本周波数を調整したい場合にも，オシレータを交換すればよい．

〔表2〕ファンクション1（FN1）のレジスタマップ

アドレス	説明	
	DLAB（LineControl[7]）=0	DLAB（LineControl[7]）=1
00000	Read/Write Buffer	Divisor[7:0]
00001	Interrupt Mask	Divisor[15:8]
00002	Interrupt Event	FIFO Control
00003	Line Control	
00004	Modem Control	
00005	Line Status	
00006	Modem Status	
00007	Scrach	
0000C	RFIFO1 Count	
1FFF0	GPIO Interrupt Mask	
1FFF1	GPIO Interrupt Event	
1FFF2	GPIO Data	
（上記以外）	（予約）	

CG100-EDKのレジスタ構成

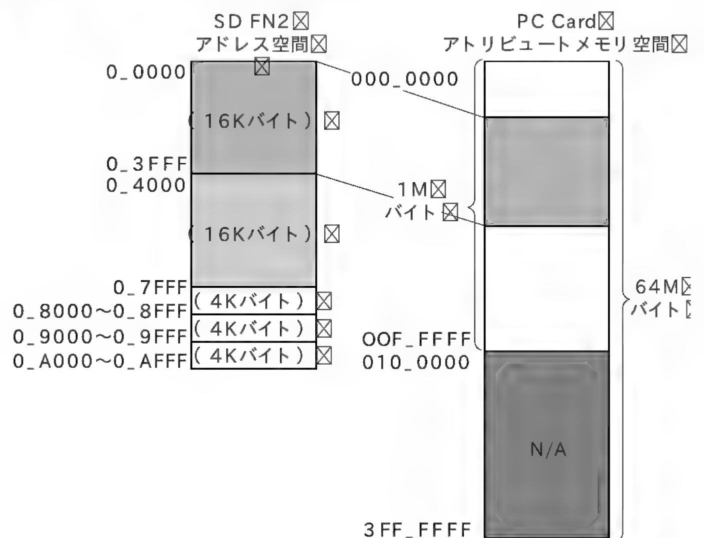
SDIOホストのソフトウェアから見えるCG100-EDK（正確にはCG100）のファンクション1（FN1）のレジスタマップを表2，ファンクション2（FN2）のレジスタマップを表3に示す．表に示さなかったが，ファンクション0（FN0）のレジスタは，I/Oファンクションの種類に関係なく，SDIO規格によって決められている．詳細は，SDIO規格書を参照されたい．ファンクション1はUARTの制御用，ファンクション2はPC Cardの制御用としてアサインされている．

GPIOは，ファンクション1とファンクション2の両方から

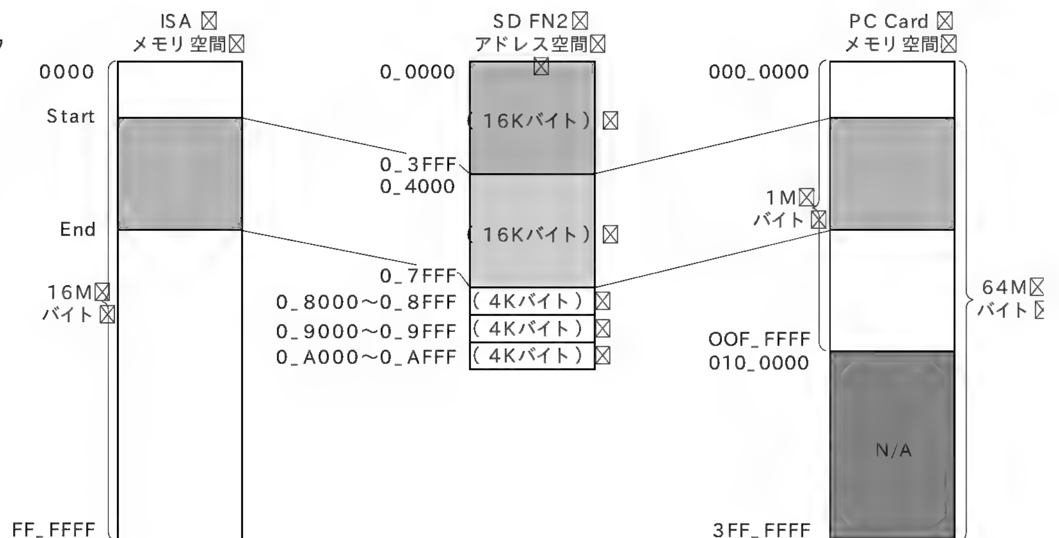
〔表3〕ファンクション2(FN2)のレジスタマップ

アドレス	説　明
00000～03FFF	Attribute Space Access Data Window
04000～07FFF	Memory Space Access Data Window
08000～08FFF	I/O Space 8bit/16bit Access Data Window
09000～09FFF	I/O Space 8bit Access Data Window
0A000～0AFFF	I/O Space 16bit Access Data Window
0C000	Attribute Space Address Offset (PC Card address[19:12]ビット)
0C100	Memory Space Address Offset (PC Card address[19:12]ビット)
0C200	I/O Space Address Offset (PC Card address[19:12]ビット)
0C300	Access bit width control
0C400	Interrupt Event
0C500	Interrupt Mask
0C600	Setup Timing Control
0C700	Command Timing Control
0C800	Recovery Timing Control
1FFF0	GPIO Interrupt Mask
1FFF1	GPIO Interrupt Event
1FFF2	GPIO Data
(上記以外)	(予約)

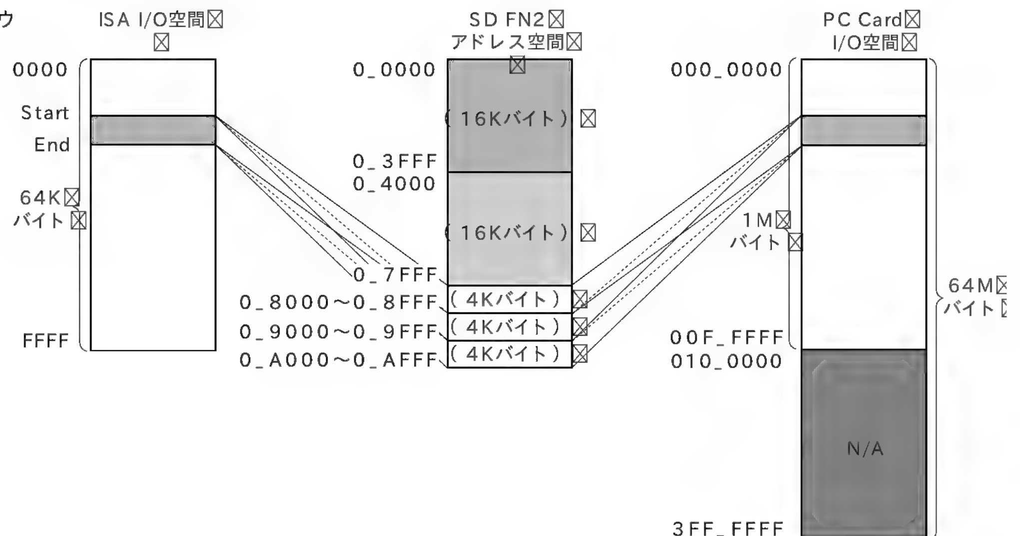
〔図9〕アトリビュートスペースのデータウィンドウ



〔図10〕メモリスペースのデータウィンドウ



〔図11〕I/Oスペースのデータウィンドウ



アクセスできるようになっていて，あるI/Oファンクションを UART に接続した場合，おもなデータ送受信はUARTで行うが，付加的な制御をGPIOを通しても行えるように配慮されている．

ファンクション 1の0_0000h番地から0_000Fh番地までは 16550チップと同じレジスタ構成になっている．1_FFF0h番地から1_FFF2番地は，GPIO制御用のレジスタである．

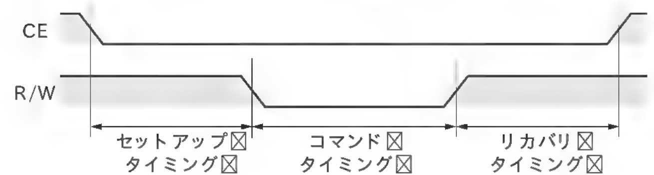
ファンクション 2の0_0000h番地から0_AFFFh番地までは，データウィンドウである．アトリビュートスペースのデータウィンドウを図9に，メモリスペースのデータウィンドウを図10に，I/Oスペースのデータウィンドウを図11に示す．

PC Cardのアドレス信号は本来26本であるが，SDコマンドの引き数で与えられるアドレスは17ビットである．その17ビットの空間に，アトリビュートスペース，メモリスペース，I/Oスペース，PC Card制御レジスタを配置しているが，各スペースをそのまま配置するのは不可能なので，データウィンドウという手法を用いている．CG100のPC Cardインターフェースに出ているアドレス信号は20本であり，そのうちの最上位8ビットは0_C000h番地，0_C100h番地，0_C200h番地の各スペースのオフセット値によって与えられる．最下位12ビットは，データウィンドウのアドレス，すなわちSDコマンドの引き数で与えられたアドレス値の下位12ビットがそのまま使用される．アトリビュートスペースとメモリスペースについては，オフセット値のビット 1～0と，SDコマンド引き数で与えられたアドレス値のビット 13～12が重なるが，それらは加算されることになる．I/Oスペースのデータウィンドウは，大きさが4Kバイトで，オフセット値を変更することでデータウィンドウが移動する単位も4Kバイトである．一方，アトリビュートスペースとメモリスペースのデータウィンドウは，大きさが16Kバイトあり，オフセット値を変更することでデータウィンドウが移動する単位はI/Oスペースと同じ4Kバイトである．

ところでPC Cardでは，そのバス上のデータ幅として8ビットの場合と16ビットの場合がある．アトリビュートスペースやメモリスペースは，接続相手によってどちらかに決まる場合が多いが，I/Oスペースは，8ビットアクセスレジスタと16ビットアクセスレジスタが混在する場合もある．そこで，0_8000h番地から0_8FFFh番地までにアクセスした場合は，PC Card制御レジスタの値によってデータ幅を決定し，0_9000h番地から0_9FFFh番地までにアクセスした場合は，8ビットデータ幅でデータ転送を行い，0_A000h番地から0_AFFFh番地までにアクセスした場合は，16ビットデータ幅でデータ転送を行うようになっている．0_C000h番地から0_CFFFh番地まではPC Card制御用のレジスタである．1_FFF0h番地から1_FFF2番地は，GPIO制御用のレジスタだが，実体はファンクション 1からアクセスされるGPIOと同じGPIOである．

PC Cardインターフェース上の信号タイミングは図12のようになる．これらのタイミングは0_C600h番地のセットアップタイミングコントロールレジスタ，0_C700h番地のコマンドタイミングコントロールレジスタ，0_C800h番地のリカバリタイミングコントロールレジスタの3種類のPC Cardタイミング調整レジスタによって調整される．

〔図12〕PC Cardインターフェース信号タイミング



システムクロックがSDIOカード内のオシレータから供給されるなど固定である場合は，電源投入後に一回だけ設定すればよい．この場合はホストからではなく，CG100の内蔵マイコンによって設定できる．システムクロックがSDバスから供給される場合は，そのクロック周波数に合わせて，PC Cardタイミング調整レジスタをホストから調整する必要がある．といっても，最高周波数の25MHzでも動作するように調整しておけば問題ないだろう．

SDIOカードのプロトタイプ設計事例

次にSDIOカードのプロトタイプ設計事例をいくつか紹介する．

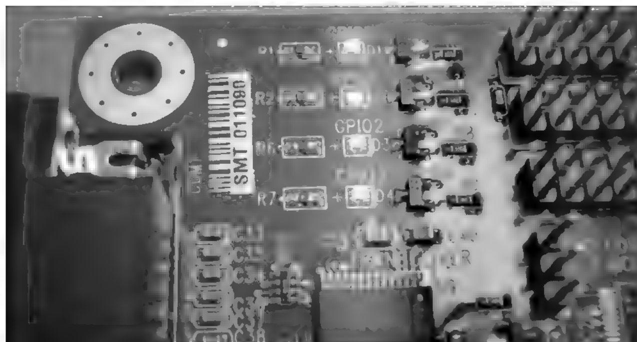
まずは数ビットのLEDを点灯制御する『SDIOインジケータカード』について紹介する．最大でも8ビットの情報のインジケータならば，CG100のGPIOを直接利用できる．それ以上の場合は，PC Cardインターフェースのアドレスバスを GPIOとして直接利用する方法があり，この場合は最大24ビットの情報のインジケータを作ることができる．さらに多くの情報のインジケータを作るには，表示情報のバッファと表示器を備えた回路をCG100のUARTインターフェースあるいはPC Cardインターフェースから制御する必要がある．

CG100-EDK上には写真4のように，GPIO3からGPIO0までの4ビットにLEDが装備されている．ホストからSDバスを通してGPIOに"1"を書くと点灯し，"0"を書くと消灯する．

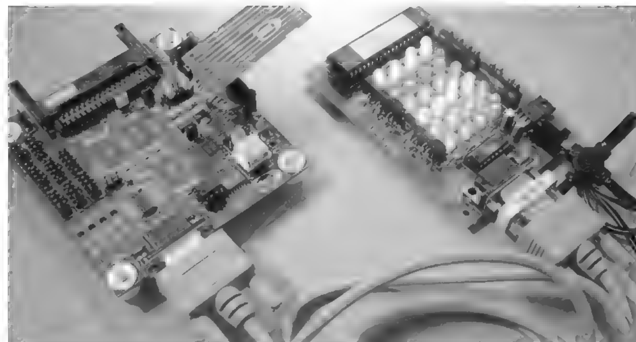
次に，UARTインターフェースを利用したSDIOカードプロトタイプを紹介する．写真5は，CG100-EDKにAVR LED(ほたる)キット(秋月電子通商)を接続した『SDIOほたるカード』である．これはAVR4414(アトメル)という8ビットプロセッサで24個のカラフルなLEDを光らせるもので，その発光パターンをUARTからコントロールできる．先に述べたように，より多くの情報のインジケータとしてUARTインターフェースを利用した例である．

写真6は，CG100-EDKにモデムを接続した『SDIOモデムカード』である．このように，既存のUARTインターフェースをもったI/Oファンクション機器をCG100-EDKに接続するこ

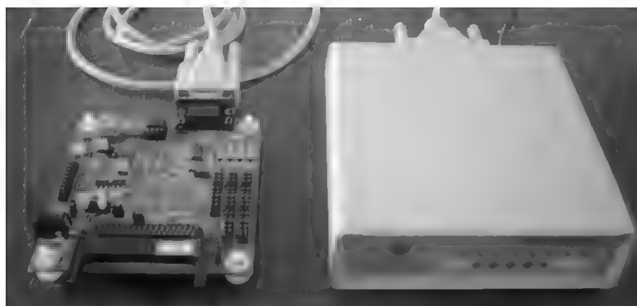
〔写真4〕SDIOインジケータカードのプロトタイプ例



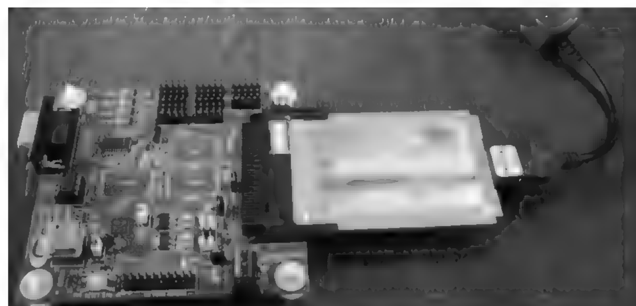
〔写真5〕SDIOほたるカードのプロトタイプ例



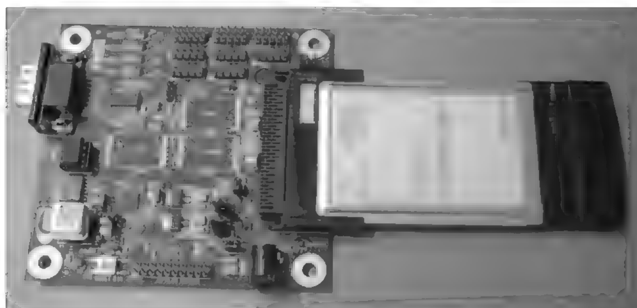
〔写真6〕SDIOモデムカードのプロトタイプ例



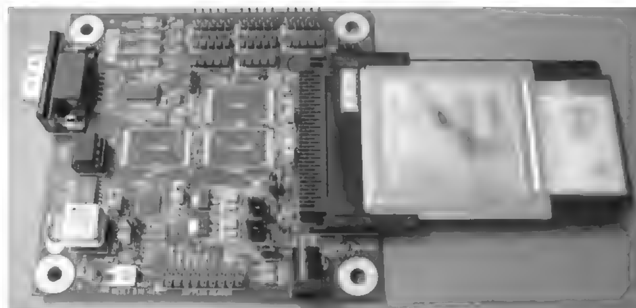
〔写真7〕SDIO有線LANカードのプロトタイプ例



〔写真8〕SDIO無線LANカードのプロトタイプ例



〔写真9〕SDIOストレージカードのプロトタイプ例



とにより，すぐにSDIOカードにすることが可能となり，ホストからSDバスを通した制御ができる．

PC Cardインターフェースを利用したSDIOカードプロトタイプ例として，写真7に『SDIO有線LANカード』を，写真8に『SDIO無線LANカード』を示す．それぞれCG100-EDKにPC Card型LANカード，およびPC Card型無線LANカードを接続している．

また，写真9は，CG100-EDKにPC Card-CFアダプタを通してCFメモリカードを接続した『SDIOストレージカード』である．小容量ではあるが，これでもストレージに見立てることができる．CFカードにマイクロドライブを使えば，より大容量のSDIOストレージカードのプロトタイプが完成する．

まとめ

以上，SDIOカードの設計アプローチの方法を概説した．また SDIOカードコントローラチップとしてはCG100を解説し，これを搭載したSDIOカード開発ボードCG100-EDKを活用した各種のSDIOカードプロトタイプの具体例を紹介した．

次回は，今回紹介したSDIOカードの中の代表的なSDIOカードの動作試験について解説する．また，ドライバ開発についても若干言及する予定である．

■SDIOカードコントローラチップCG100およびSDIO開発環境ツールSD-IDEの問い合わせ先
シイガイズ（株）
E-mail：Sales@c-guys.com

やそしま・ひろゆき　シイガイズ（株）　技術開発部　Senior SOC Engineer

TOPPERSで学ぶ RTOS技術

第4回 サービスコールの概要・その1

岸田 昌巳

今回は連載第2回目(2003年10月号)で解説したシミュレーション環境を使いながら各サービスコールについて説明します．Windows上でのシミュレーション環境を使うので，Microsoft Visual C++ 6.0/Visual BASIC 6.0が必要となります．

また，サービスコールの説明ということもあり，初心者でも理解できるよう，μITRON4.0仕様を網羅した全般的な解説になっています．

μITRONの特徴

組み込み用RTOSとして大きなシェアを持つμITRON仕様は以下の設計方針に基づいています．仕様書に明記されているので，ご存じの方も多いでしょう．

1) ハードウェアを過度に仮想化することを避け，ハードウェアに対する適応化を考慮する
2) アプリケーションに対する適応化を考慮する
3) ソフトウェア技術者の教育を重視する
4) 仕様のシリーズ化やレベル分けを行う
5) 豊富な機能を提供する

この5項目に関しては前回までの解説やμITRON4.0仕様書に詳細な説明があるので，詳しくはそちらを見てください．

ここでは初心者が理解するために必要な部分に注目してみましょう．一貫性のある用語の使い方，名称の付け方は，理解を深めるためにも，知識として蓄積するためにも効果があります．サービスコールの名前の付け方に関しては表1を参照してください．

これらを覚えておくだけで，サービスコールが類推できるようになります．このわかりやすさもITRONの特徴です．

タスク関連のサービスコール

サービスコールの説明に入る前に少し触れておきたい点を述べます．

● タスク分割に関して

システムの概略設計段階や構成をデザインする段階で，どのような方針でタスクを分割するかを考えると思います．

この検討の中で，タスクをいくつ用意するのかが決まります．具体的な分割の方法は，システム全体で提供している機能を，機能ごとに分割する方法や，デッドラインをもとに分割する方法など複数あります．

また逆に，いったんは機能別に分割したタスクを，性能を上げるために再度結合することもあります．結合する理由としては，全体のふるまいから必要ないタスクスイッチなどが減るように処理のむだを省く，いわゆるチューニングにともなう変更の事例が多いようです(図1)．

ここでは，対象とする組み込みシステムをタスクごとに分割する方法として，機能分割を行ったと仮定します(チューニングに関しては当面，横に置いておく)．

● 機能分割について

機能別に対象とする組み込みシステムを分割すると，すべてのサブシステムが，入力や，データ加工，出力の機能をもっていることがわかります．例外なくほとんどのシステムで当てはまると思います．

機能分割ではこれらの入力や，データ加工，出力のそれぞれをタスクに割り当てます．

タスク関連のサービスコールでは，このタスクを動かす，止めるといったタスクを直接制御するサービスコール(タスク管理機能)と，簡単な同期機能を取るためのサービスコール(タスク付属同期機能)，例外処理のサービスコール(タスク例外処理機能)を提供しています．ここではこれらサービスコールについて解説します．

● タスク管理機能

タスクを管理するための関連するサービスコールは，大まかにはxxx_tskとあるサービスコール(たとえばact_tskなど)と，このxxxの部分をもつyyy_xxx(can_actなど)があります．表1の略号一覧をもとにしてxxxの部分にいろいろ当てはめてみるとわかるかと思います．

TOPPERS/JSPでは，タスク関連のサービスコールとして，タスク管理機能，タスク付属同期機能，タスク例外処理機能をもちます．これらは全部で表2に示すだけのサービスコールがあります．

〔表1〕サービスコールの略号一覧

表記	元の意味	解　説
acp	Accept	受理
act	Activate	活性化する
alm	Alarm Handler	アラームハンドラ
att	Attach	付随する
blf	Fixed-size Memory Block	固定長メモリブロック
blk	Memory Block	可変長メモリブロック
cal	Call	呼ぶ
can	Cancel	キャンセル
chg	Change	変更
cpu	CPU	CPU
cre	Create	生成
cyc	Cyclic Handler	周期起動ハンドラ
dat	Data	データ
def	Define	定義
del	Delete	削除
dis	Disable	禁止
dly	Delay	遅延
dsp	Dispatch	切り替え
ena	Enable	許可
exd	Exit and Delete	出口（終了）と削除
ext	Exit	出口（終了）
fwd	Forward	前に
flg	Eventflag	イベントフラグ
get	Get	取得
int	Interrupt, Interrupt Handler	割り込みハンドラ
loc	Lock	ロック
mbf	MessageBuffer	メッセージバッファ
mbx	Mailbox	メールボックス
mpf	Fixed-size MemoryPool	固定長メモリプール
mpl	MemoryPool	可変長メモリプール
msg	Message	メッセージ
pol	Poll	ポーリング
por	Port	ポート
ras	Raise	上げる
rcv	Receive	受信
rdv	Rendezvous	集合
ref	Refer	参照
rel	Release	解放
ret	Return	戻る
rot	Rotate	回転
rsm	Resume	再開
sem	Semaphore	セマフォ
set	Set	設定
sig	Signal	シグナル
slp	Sleep	寝る（起床待ち）
snd	Send	送信
sns	Sense	感覚（センス）
sta	Start	スタート
stp	Stop	ストップ
sus	Suspend	サスペンド
ter	Terminate	終了
tim	Time	時刻
tsk	Task	タスク
unl	Unlock	アンロック
ver	Version Information	バージョン
wai	Wait（also used in yyy form）	待ち
wup	Wakeup（also used in yyy form）	起床

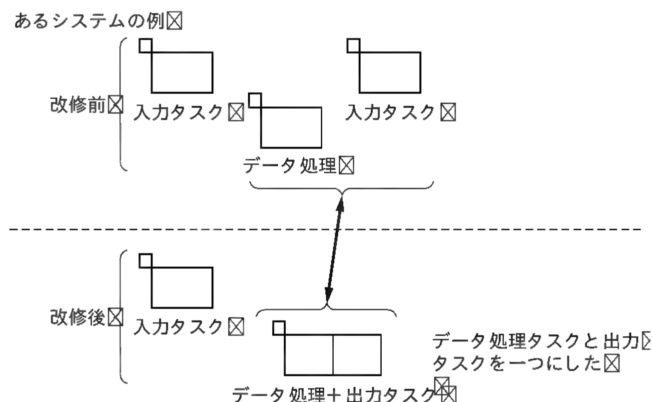
μITRON4.0仕様書の記述に追記している．

● 最初の一歩

理解するためには実際に動かしてみるのが一番です．今回はシミュレーション環境でサンプルを動かしつつ説明します．

まずはじめに，以下のサンプルを，いろいろ変えてみながら動かしてみるのがわかりやすいでしょう．ここでは実際に元のファイルをいろいろ変更して理解を深めたいと思います．

〔図1〕タスク分割，結合



〔表2〕タスク管理機能

タスク管理機能	
CRE_TSK	静的API
ER act_tsk(ID tskid);	タスク起動
ER iact_tsk(ID tskid);	タスク起動
ER_UINT can_act(ID tskid);	タスク起動要求のキャンセル
void ext_tsk(void);	タスクの終了
ER ter_tsk(ID tskid);	タスクの強制終了
ER chg_pri(ID tskid, PRI tskpri);	タスク優先度の変更
ER get_pri(ID tskid, PRI *p_tskpri);	タスク優先度の参照
タスク付属同期機能	
ER slp_tsk(void);	起床待ち
ER tslp_tsk(TMO tmout);	起床待ち（タイムアウトあり）
ER wup_tsk(ID tskid);	タスクの起床
ER iwup_tsk(ID tskid);	タスクの起床
ER_UINT can_wup(ID tskid);	タスクの起床要求を無効化
ER rel_wai(ID tskid);	待ち状態の強制解除
ER irel_wai(ID tskid);	待ち状態の強制解除
ER sus_tsk(ID tskid);	強制待ち状態への移行
ER rsm_tsk(ID tskid);	強制待ち状態からの再開
ER frsm_tsk(ID tskid);	強制待ち状態で再開
ER dly_tsk(RELTIM dlytim);	自タスクの遅延
タスク例外処理機能	
DEF_TEX	静的API
ER ras_tex(ID tskid, TEXPTN rasptn);	タスク例外処理の要求
ER iras_tex(ID tskid, TEXPTN rasptn);	タスク例外処理の要求
ER dis_tex(void);	タスク例外処理の禁止
ER ena_tex(void);	タスク例外処理の許可
BOOL sns_tex(void);	タスク例外処理禁止状態の参照

シミュレーションの環境では，JSPカーネル配布物以外に以下のファイルを用意します(これらは連載2回目のファイルから一部変更した)．

● 修正の方法

連載2回目のソースを変更しました．変更点は表3の部分です．サンプルから何か試作される場合の修正もほぼ同じ部分を修正することになると思われます．

今回，ファイル名だけの修正なので，拡張子.dspと.dswファイルは大きく変えていませんが，ディレクトリの位置関係を変えた場合などは，この二つのファイルも修正してください(リスト1)．

ここから先は設計に合わせてこのソースを適宜変更しながら，サービスコールの使用法の説明を行います．

開発の流れ

開発は，大体，以下のような流れになります．今回は図2の囲んだ部分が対象になります．サービスコールの解説は詳細設計やコーディングのときに役立ちます．書籍や所属する会社などの違いから，この囲んだ部分でさえ，各社まちまちの場合もありますが，たいていは詳細設計であったりコーディング，デバッグです．

〔表3〕修正個所

sample1.c	タスクを記述：タスクを増やして中身を修正．ファイル名も変更
sample1.h	定数定義部分：定数定義部分を修正．ファイル名も変更
sample1.cfg	静的APIを用いたタスクの登録：タスクを追加
Toppers.dsp	VisualStudioのIDE設定ファイル(nmake用Makefile)：ターゲットファイル名の変更
Toppers.dsw	VisualStudioのIDE設定ファイル(Workspace File)：今回は修正なし
window.chk	チェックファイル：今回は修正なし

〔リスト1〕ソースの変更点

```
●sample1.c
input_taskタスク，output_taskタスクの記述を行います.

●sample1.h
2c2
<  * CQ出版  Interface 2003年10月号
---
>  * CQ出版  Interface 2003年2月号
49c49,50
<extern void task(VP_INT tskno);
---
>extern void input_task(VP_INT tskno);
>extern void output_task(VP_INT tskno);

●sample1.cfg
2c2
<  * CQ出版  Interface 2003年10月号
---
>  * CQ出版  Interface 2003年2月号
13c13,14
<CRE_TSK(TASK_ID, {TA_HLNG, (VP_INT)1,task,MID_PRIORITY,
                                        STACK_SIZE,NULL});
---
>CRE_TSK(TASK_ID1, {TA_HLNG, (VP_INT)1,input_task,MID_PRIORITY,
                                        STACK_SIZE,NULL});
>CRE_TSK(TASK_ID2, {TA_HLNG, (VP_INT)1,output_task,MID_PRIORITY,
                                        STACK_SIZE,NULL});
```

● 作成するサンプルの前提条件

機能分割の説明で，入力，データ処理，出力と三つに分けると説明したので，これら三つのタスクの起動方法，タスク間の連携方法を複数のサービスコールで解説します．ただし，前半部分では複数のタスクが絡む話は少ないので，一つのタスクを取り出して説明します．

● タスクの実態，生成までの流れ

TOPPERS/JSPだけに限らないのですが，マルチタスクシステムにおけるタスクの実体は，単なる関数です．この関数を登録することで，タスクとして扱うことができます．

先ほど述べたように，システムを機能ごとにサブシステムに分割し，そのサブシステムの各部分ごとに，タスクを割り当てるように設計します．μITRON4.0のスタンダードプロファイルでは，このタスクを作るためにコンフィギュレーションファイルに登録します．

TOPPERS/JSPはスタンダードプロファイルにしたがって実装されているので，同じくコンフィギュレーションファイルで登録します．このコンフィギュレーションファイルでのタスク登録には静的APIを利用します．使用する静的APIは，タスク生成の静的APIであるCRE_TSKです．この静的APIにタスクとして登録したい関数へのポインタをパラメータとして渡します．

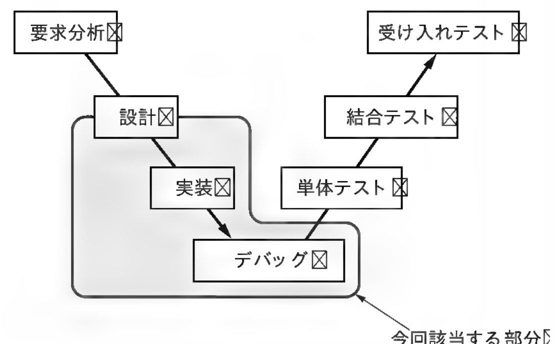
● CRE_TSK：タスクの生成の実際

生成したタスクの状態は以下のように変化します．以下はカーネル内部の状態管理の処理からみた動きです．

1) 生成したタスクを，まず休止状態(DORMANT)にする
2) act_tskサービスコールが発行されると，指定したタスクを実行可能状態(READY)にする
3) 設定された優先順位にしたがって，一番優先順位が高かったタスクを実行状態(RUNNING)にする
4) タスクへ制御を移す．これによりタスク内の処理が実行される

実際に登録の処理を見てみましょう(リスト2)．ここでは入

〔図2〕Vモデル



〔リスト 2〕タスク登録（t4-ex01¥sample1.cfg の 14行目）

```
/*三つのタスクを登録*/
CRE_TSK(TASK_ID1, {TA_HLNG, (VP_INT)1,input_task,MID_PRIORITY,
                                                   STACK_SIZE,NULL});
CRE_TSK(TASK_ID2, {TA_HLNG, (VP_INT)1,output_task,MID_PRIORITY,
                                                   STACK_SIZE,NULL});

CRE_TSK(MAIN_TASK, {TA_HLNG|TA_ACT,0,main_task,MAIN_PRIORITY,
                                                   STACK_SIZE,NULL})
```

力側タスクを input_task，出力側を output_task とします．

順にパラメータを説明すると，TASK_ID1 はタスクの識別用の ID です．TA_HLNG は高級言語で記述していることを示します．(VP_INT)1 はタスクに引き渡す拡張情報です．関数の引き数として渡されます．

あとは，タスクとして登録する関数名，優先度，与えるスタックのサイズ，スタック領域の先頭位置を示します．スタックの先頭位置が NULL なので，TOPPERS/JSP カーネル側でスタックを用意します．

ちなみに，スタンダードプロファイルでは，スタック先頭位置はかならず NULL となり，TA_HLNG 以外をサポートしません．

● タスクの状態（図 3）

ITRON4.0仕様では，タスクを生成すると，何もない状態，未登録状態（NON-EXISTENT）から休止状態（DORMANT）になります．タスクは存在するが動いていないので休止といいます．この休止状態は，待ち状態（WAITING）とは違い，何かの待ち要因を待っているわけではありません．

なお，休止状態のタスクは，実行状態（RUNNING）になったことがなく，一度も実行されていません．タスク内部の初期化処理，つまり登録した関数内の初期化処理も実行されていません．待ち状態のタスクは，一度は実行状態にあったタスクがサービスコールを発行することで待ち状態に入っているので，登録した関数内の初期化処理は実行されています．

このため，タスクの起動後，最初に行う処理と 2回目以降に行う処理とにかかる時間が大きく違う場合があります．この初期化処理にかかる時間は注意が必要です．これは状態遷移図上の実行状態から休止状態に，再度入った場合も同じです．

● 実行可能状態

休止状態から実行可能状態にするにはサービスコールを使用するか，タスク生成時のオプション TA_ACT を用いることで最初から実行可能状態にすることができます．TA_ACT をすべてのタスクに指定した場合，優先順位の高いタスクから実行されます．

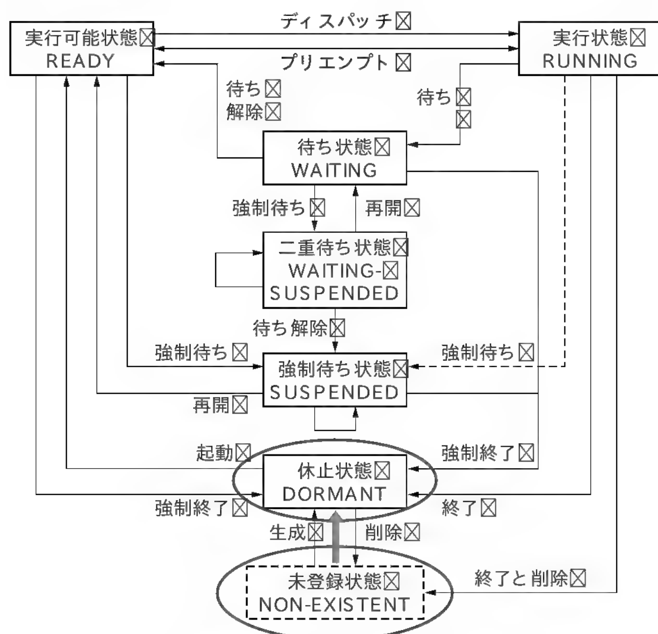
タスクの初期化処理をある順序にしたがって順に行いたい場合は，登録したタスクのうち，どれか一つは TA_ACT を指定し，TA_ACT を指定したタスクから他のタスクを起動するような方法もあります．

なお，ここでは，main_task が input_task，output_task を起動します．前述のタスク，main_task から初期化を始める方法では，内側から外側に向かって初期化を進めることになります．これは周辺デバイスの初期化と同じで，初期化処理のセオリにも合致しています．

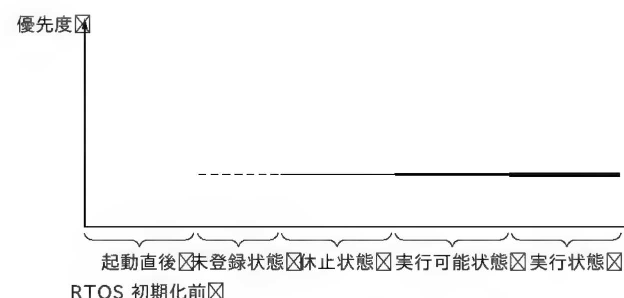
● コンパイル

サンプルコードから変更を行い，実際に動かしてみるためにはコンパイルが必要です．サービスコールの説明に入る前に簡単におさらいをしておきましょう．

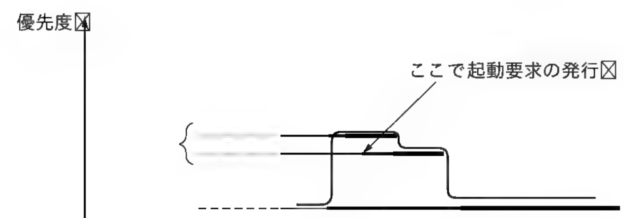
〔図 3〕タスク生成



（a）タスク状態遷移図：何もない状態から DORMANT



（b）タスクの実行履歴：点線から実線に



（c）タスクの実行履歴（凡例）：点線から実線に

1) 複数のタスクからなるアプリケーションを作成する場合，sample.cfgなどコンフィギュレーションファイルに生成したいタスクの情報を追記する．ここにはタスクにしたい関数名などを記述する．
2) 定義したタスクの処理本体である関数をsample.cなどのC言語のソースファイルに記述する．プロトタイプはsample.hに記述すること．
3) コンパイルはMakefileに記述があるように，コンフィギュレータがsample.cfgファイルからkernel_cfg.c，kernel_chk.c，kernel_id.hを生成する．このソースを修正する必要はない．
4) ソース，設定ファイルの修正が終わった後，コンパイルする．
5) 最初のコンパイルではJSPのカーネル部分もアプリケーション部分もあわせて，すべてをコンパイルする．

〔表4〕エラーコード

エラーコード	内　容
E_SYS	システムエラー（内部エラー）
E_NOSPT	未サポート機能（提供されていない機能を指定した）
E_RSFN	予約機能コード（提供されていない機能を指定した）
E_RSATR	予約属性（提供されていない属性を指定した）
E_PAR	パラメータエラー（パラメータ値が不正である）
E_ID	不正ID番号（オブジェクトのIDが範囲外である）
E_CTX	コンテキストエラー（呼び出せないコンテキストから呼び出した）
E_MACV	メモリアクセス違反（アクセスしてはいけないメモリ番地をアクセスした）
E_OACV	オブジェクトアクセス違反（アクセスしてはいけないオブジェクトを使った）
E_ILUSE	サービスコール不正使用
E_NOMEM	メモリ不足
E_NOID	ID番号不足
E_OBJ	オブジェクト状態エラー
E_NOEXS	オブジェクト未生成
E_QOVR	キューイングオーバフロー
E_RLWAI	待ち状態の強制解除
E_TMOUT	ポーリング失敗またはタイムアウト
E_DLT	待ちオブジェクトの削除
E_CLS	待ちオブジェクトの状態変化
E_WBLK	ノンブロッキング受け付け
E_BOVR	バッファオーバフロー

その後，各タスクの中身や処理内容に関わる修正をした場合は，make toolを使用していることから，JSPのカーネル部分のコンパイルは行われず，修正したソースのみをコンパイルできるようになります．

ちなみにコンフィギュレータの役割が理解できていないと，1)でのタスクのIDなどの扱いがよくわからないのではないかと思います．このあたりを簡単に補足しておきます．

例としてタスクの追加を取り上げます．リスト2に対して，もう一つTASK_ID3などとして追加したい場合は，sample.cfgに記述されているタスクの定義を参考に，IDをTASK_ID3のように記述して追記してください．

この記述を元にコンフィギュレータはIDを定義したファイルを生成します．このため，他のファイルにTASK_ID3などの定義を行う必要はありません．このTASK_ID3の定義，何の値が割り振られるかはコンフィギュレータで決定されます．決まった値はkernel_id.hに定義されます．

なお，関数のプロトタイプ宣言はsample.hに記述しておきます．このプロトタイプ宣言は生成したファイル内で参照されています．

サンプルソースをコンパイルした後，kernel_cfg.c，kernel_chk.c，kernel_id.hに記述されている内容を見るとわかりやすいかもしれません．

サービスコールの解説

ここからは各サービスコールごとに説明を行います．リターンパラメータ（返り値）は，現在のTOPPERS/JSPで返す値と返さない値も記述し，返さない値に関してはカッコ内に記しています．なお，正常終了の場合はE_OKが返ります．

エラーコードには表4に示すだけの種類があります．

ちなみに，後ほど出てくる，このカッコありの返り値のほとんどの場合が致命的なエラーです．エラー処理を省略して良いことはありません．カッコありの場合についても互換性や今後の機能拡張など，先を考えてエラー処理の記述を行っておくことを勧めます．

タスクの起動

C言語API
```
ER act_tsk(ID tskid);
ER iact_tsk(ID tskid);
```
パラメータ

ID tskid;　タスクID番号

リターンパラメータ

ER E_OK（正常終了）またはエラーコード

エラーコード

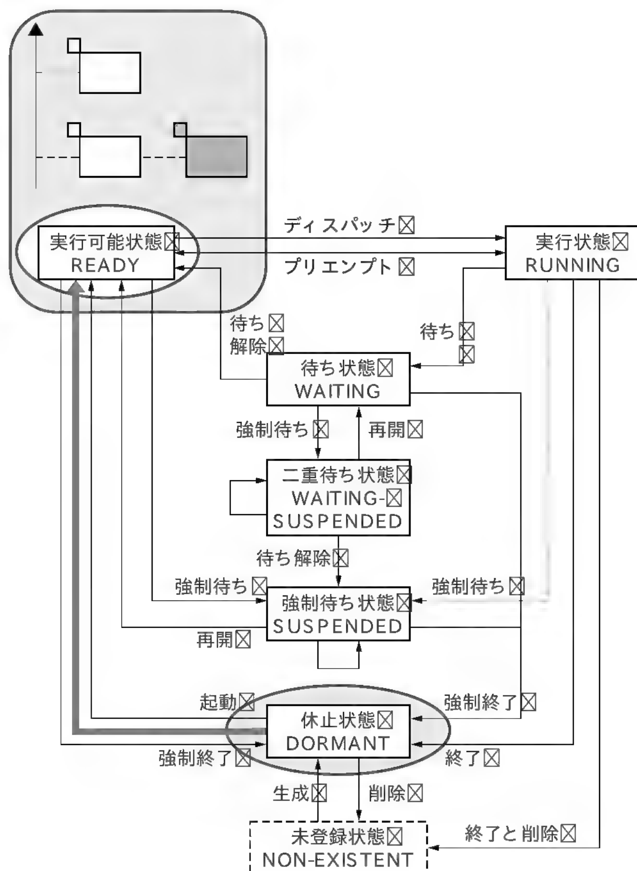
(E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_QOVR

E_CTX ：コンテキストエラー（呼び出せないコンテキストから呼び出した）
E_ID ：不正ID番号
E_QOVR：キューイングオーバフロー

指定したタスクを休止状態から実行可能状態にします（図4）．実行中にあるタスクの状態を実行状態と言いますが，他の状態と違い，この状態にあるタスクは一つしかありません．

頭にiの付くサービスコールiact_tskはタスク以外の場所，ハンドラ内や拡張サービスルーチン内で使用します．

〔図4〕DORMANTからREADYへ



現在実行中のタスクより優先順位の高いタスクに起動がかけられた場合，タスクスイッチが発生し，より優先順位の高いタスクに切り替わります．優先順位の高いタスクが実行状態になり，今まで実行状態にいたタスクは実行可能状態に状態遷移し，いったんタスクの実行が止められます．いったん止められる理由は優先順位の高いタスクが実行されているからで，優先順位の高いタスクがいなくなれば，再度実行状態に入り，タスクは止められた所から続きを実行します．

優先順位の低いタスクに起動がかけられた場合は，優先順位の低いタスクは実行可能状態に入るだけで，実行中のタスクに変化はありません．

● 起動要求に関する補足

act_tskの起動要求はキューイングされますが，TOPPERS/JSPの実装ではカウンタで管理されているのではなく，1ビットで管理されています．このため，1回分しかキューイングされません．二つ以上起動要求がかかるとキューイングのオーバフローが発生します．

スタンダードプロファイルでは，起動要求1以上でいくつあっても良いとなっているので，仕様上，最低限の機能を実装しているといえます．

なお，キューイング可能な個数が多いからといって，それが良いカーネルだとは限りません．

● キューイング

キューとは行列を意味しており，先に到着したものから順に処理されるしくみを作るなどに使われます．

たとえば，イベントやメッセージなどをやりとりする場面では，送り側と受け側の間にキューと呼ぶ領域を設け，受け側が処理できない場合に，待たせるための機能として用意します．ここでのイベントなど，やりとりする実体は，入った順に処理されます．この行列に関わる処理をキューイング（Queuing）と呼びます．キューイングされた要求の処理は，実施できる状態である場合に実施され，実施できない場合は待たされます．

なお，キューにノードを加える処理をエンキュー（entry-queue→en-que），キューからノードを外す処理をデキュー（delete-queue→de-que）と呼ぶこともあります．

● ネスト

ネストは入れ子を意味します．μITRON仕様では，タスクの状態に関するネストが，サービスコールを発行することで発生します．

サービスコールの発行により，ある状態から別の状態へ遷移するのですが，これを元に戻すには，遷移した順序と逆順で戻る必要があります．そのため，これをネストと呼んでいます．

たとえば，後述する待ち状態から強制待ち状態に入るなど，再度別の状態に入った場合，強制待ちを解除しても実行状態に

〔リスト3〕タスクの起動のサンプル

```
void control_task(VP_INT exinf)
{
    act_tsk(TASK_ID1);  /* 入力タスクを起動(READY状態へ) */
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "input task起動");
    proc_something();
    ext_tsk();          /* タスク終了(DORMANT状態へ) */
    syslog_0(LOG_NOTICE, "ここには来ません");
}
//-----------------------------------------------------------
```

```
void control_task(VP_INT exinf)
{
    :
    /* 入力タスクを起動(READY状態へ) */
    sta_tsk(TASK_ID1,extinf-evn);
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf) // <---exifにexinf-evnが渡される
{
    syslog_0(LOG_NOTICE, "input task起動");
    proc_something();
    ext_tsk();          /* タスク終了(DORMANT状態へ) */
    syslog_0(LOG_NOTICE, "ここには来ません");
}
```

は戻れず，待ち状態にしかなりません．つまり，ネストされた要求は，解除処理を行わないと元の状態に戻れません．

キューイングが実行できる状態になった時点で実施されるのと違い，プログラム上にサービスコールを使って，ネストさせる要求を出す処理と，それを解除する処理を記述しておく必要があります．

● sta_tsk の代替としての act_tsk：JSP の範囲外について

μITRON4.0仕様には，起動をかけるサービスコールとして act_tsk 以外に sta_tsk が存在します注1．

sta_tsk と act_tsk の違いは，タスクに対して，タスクを起動するときの「タスクの起動コード」を引き渡すことができるかどうかです．また，sta_tsk は休止状態のタスクに対してのみ発行でき，他の状態ではキューイングせず，未登録状態のときはエラーとなります．

ただ，もし sta_tsk で行うことが，起動要求の処理だけであるならば，act_tsk でも十分に代用可能です．sta_tsk を利用したソフトウェア部品を TOPPERS/JSP で利用する場合に，sta_tsk がないからと諦める必要はないでしょう．

タスクの起動要求のキャンセル

C言語API
ER_UINT can_act(ID tskid);
パラメータ
ID tskid　　タスクID番号
リターンパラメータ
ER_UINT 起動要求回数またはエラーコード
エラーコード
(E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS)

指定したタスクに対する起動要求をキャンセルします．キューに入っている起動要求もキャンセルされ，この時点の起動要求の回数が返り値となります．

実際には，休止状態のタスクに対し，複数の act_tsk を発行した後に can_act を発行した場合，act_tsk を発行した時点で休止状態のタスクは実行可能状態になっているので，二つ目以降の act_tsk による起動要求はキューに入ります．can_act は，このキューに入っている要求をキャンセルします．実行可能状態から休止状態にタスクが変わるわけではありません．

そのため，一つだけ act_tsk を発行した後に can_act を発行した場合には，何も起こりません．

実行可能状態にあるタスクに対し，一つだけ act_tsk を発行した後に can_act を発行した場合は，この起動要求はキューに入れられるので，このキューに入っている要求がキャンセルされます．これは複数 act_tsk を発行しても同じくキューに入っている要求がキャンセルされます．

ここで補足説明としてリターンパラメータの部分に注目してみてください．エラーコードと起動要求の回数が同じリターンパラメータで返ってきます．これは ITRON 仕様ではエラーコードはすべて負の数になっていることを利用しています．μITRON4.0仕様ではこの can_act 以外にも，can_wup など，同様な方法を使って回数などを返しているサービスコールがあります．

自タスクの終了

C言語API
void　ext_tsk(void);
パラメータ
なし
リターンパラメータ
なし

実行状態でこのサービスコールを発行すると，自タスクは休止状態になります（図5）．このサービスコールからは戻ってきません．

〔図5〕状態遷移図3．READY から DORMANT へ

注1：μITRON4.0仕様では機能セットとしてプロファイルという概念が導入された．TOPPERS/JSP はスタンダードプロファイルの範囲を実装している．ただ，この範ちゅうに sta_tsk はなく，TOPPERS/JSP に sta_tsk は存在しない．また，未登録状態というタスクの状態も TOPPERS/JSP では存在しない状態である．

● act_tsk，ext_tsk の組み合わせ

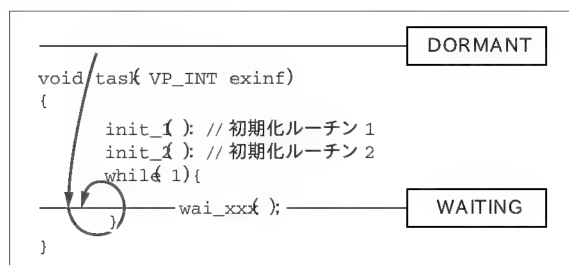
タスクで効率良く目的の処理を行うという面から見た場合，処理ごとに毎回タスクの起動と終了を行うと，タスク内の初期化部分を毎回実行することになり，効率が良くありません．毎回実行しないようにすれば，問題は発生しないでしょう（図6）．

ここまでで act_tsk，ext_tsk の組み合わせは，効率は良くないものの，簡単にタスクを制御できることがわかってもらえたと思います（リスト4，図7）．

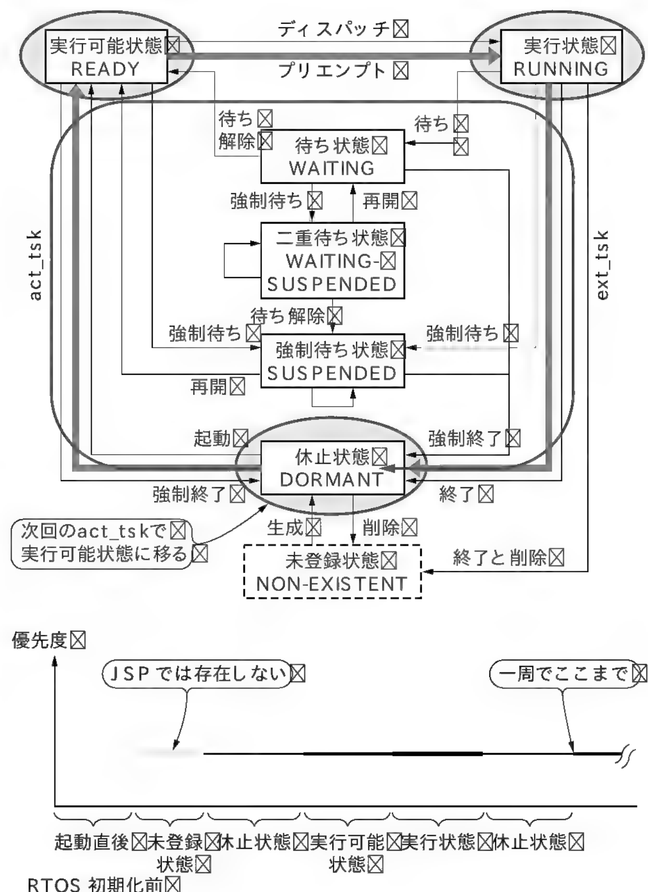
たとえば，処理自体に関連が少ない場合や相手のタスクに依存する部分が少ないときには簡単に制御できて便利です．ただし，もう少し複雑な処理をさせる場合や，これから機能拡張する予定がある場合には，最初から別のサービスコールを使用した構成を薦めます．

また，自タスクの終了に関して注意すべき点を挙げておきます．

〔図6〕プログラムの流れ



〔図7〕リスト4の動作



● 終了に関して

たとえば，ある処理をして ext_tsk を発行し，タスクを休止状態にするタスクがあったとすると，一度の起動要求のみだった場合，タスクは ext_tsk で休止状態に移行します．タスクの ext_tsk 以降の部分は実行されません（リスト5）．複数の起動要求があったとしても「ここには来ません」のメッセージを見ることはできません．

● 起動要求のバッファリング

リスト5では iact_tsk を利用しています．

iact_tsk 発行から，ext_tsk のサービスコール発行までに再度 iact_tsk が発行されると，起動要求がバッファリングされます．これは，タスクの処理中に次の割り込み処理がかかったなど，割り込み処理が複数発生した場合や，タスクの処理が予定より時間がかかり，次の割り込み処理に被ってしまった場合に該当します．

たとえば，何かのスイッチを入れると割り込みが発生する機器を考えてみましょう．押したタイミングで割り込みが入ることになっているとすると，最初のスイッチ入力が終わってから処理が終わる前までに，次のスイッチ入力が発生することもあります．このような場合に起動要求がバッファリングされます．

● 一度だけで終わりたい

次に，ある期間内に一度だけ処理を行いたい場合について考えてみましょう．

処理中に入った二つ目の実行を行いたくない場合には注意が必要です．たとえば，ext_tsk を while 文で繰り返し呼び出しても効果はありません（リスト6）．ext_tsk を呼び出した時点で，そのタスクは休止状態に入るためです．そして再度起動要求がかかった場合は，先頭から実行されます．複数回起動要求がかかった場合も，ext_tsk から抜けてくるのではなく，先頭から実行されます．

〔リスト4〕act_tsk，ext_tsk のサンプル（t4-ex04）

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    iact_tsk(TASK_ID1); /* 入力タスクを起動(READY 状態へ) */
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "input task起動");
    proc_something();
    ext_tsk();          /* タスク終了(DORMANT 状態へ) */
    syslog_0(LOG_NOTICE, "ここには来ません");
}
```

ちなみに，ある処理期間中に要望された内容を無視して，処理終了後はすぐに最初から実行されないようにするためには，can_actを呼び出した後，すぐにext_tskを呼び出せば良いように思えますが，これではcan_actとext_tskの間に割り込みが発生してiact_tskが呼び出された場合にうまくいきません（リスト7）．

● キューイング

起動要求はキューイングされており，act_tskではオーバフローしない範囲で，起動要求のカウントをインクリメントします．逆にext_tskはデクリメントします．TOPPERS/JSPではビットフィールドで定義された1ビット分のキューしか持っていないので，2回以上要求するとオーバフローします．オーバフローする場合，たいていは処理が追いついていません．

どこでいつ発生するかを押さえることは，それなりに難しいため，事前に設計された起動要求の回数などに注意し，発生した場合には普段と違うところを探す必要があります．

ほかにも最大負荷時に，起動要求が最大どの程度までカウントアップされるのかも把握しておく必要があります．この最大負荷時に予測したカウント数より多い場合も，処理が追いついていないことを示しています．

キューイングなどに関する機能拡張に興味があるのであれば，以下のWebページを参考にしてみるとよいでしょう．これは豊橋技術科学大学の若林氏のWebページです．

URL： http://www.ertl.jp/~takayuki/jsp/customize/

● エラー処理に関して：E_QOVR

エラー処理は実際のシステムで必要な処理です．あるべき姿から，どのように実現するべきかといった考察が必要です．実際に構築しようとしているシステムによって処理自体もいろいろ違うとは思いますが，ほかの文献などでも，例示自体があまり行われていないため，ここでは簡単に説明したいと思います．

たとえば，iact_tskでタスクを起動する場合，iact_tskのエラー処理が必要となります．iact_tskがエラーを返すのは，過負荷がかかった場合など，起動要求が出されているにもかかわらず処理ができないような状況を示しています．エラー処理は，できなかった処理のリカバリや処理全体を管理しているタスク，操作者への通知のために必要になります．

もう少し具体例を挙げます．スイッチを押すと割り込みが発生し，あるタスクを起動するような処理を考えてみましょう．通常は一つ一つiact_tskでタスクの起動要求を行い，この起動要求がかかるたびに起こされたタスクで何らかの処理を行うシステムがあるとします．このシステムに過負荷がかかった場合は，複数の起動要求に対する処理が追いつかず，iact_tskのキューイングが追いつかなくなり，バッファオーバフローが発生します．

TOPPERS/JSPでは，休止状態にあるタスクに対して続けざまに起動要求を出すと，一つ目でタスクが起動され，二つ目は

〔リスト5〕起動要求（t4-ex5¥sample1.c）

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    iact_tsk(TASK_ID1);      /* タスクを起動(READY状態へ) */
}

/*
 *  起動されるタスク  タスクIDはTASK_ID
 */
void input_task(VP_INT exinf)
{
    while(1){
        proc_something();
        ext_tsk();           /* タスク終了(DORMANT状態へ) */
        message("ここには来ません");
    }
}
```

〔リスト6〕終わり方

```
/*
 *  処理終了後，必ず終了したいタスク
 */
void input_task(VP_INT exinf)
{
    while(1){
        proc_something();
        while(1){        /* ループさせる意味がない */
            ext_tsk();   /* 必ずタスク終了する(DORMANT状態へ) */
        }
        message("ここには来ません");
    }
}
/*
 *  処理終了後，必ず終了したいタスク
 */
void task(VP_INT exinf)
{
    while(1){
        proc_something();
        ext_tsk();        /* 必ずタスク終了する(DORMANT状態へ) */
        message("ここには来ません");
    }
}
```

〔リスト7〕かわいそうなタイミング

```
/*
 *  タイミングによっては終了できないタスク
 */
void input_task(VP_INT exinf)
{
    while(1){
        proc_something();
        can_act(TSK_SELF)
         /* ここで割り込みが発生すると終了後，
            再度頭から実行される */
        ext_tsk();           /* タスク終了(DORMANT状態へ) */
        message("ここには来ません");
    }
}
```

キューイングされます．三つ目の要求はタスクの処理が終わらず，ext_tskが呼ばれる前であれば，バッファオーバフローが発生します．このため，バッファオーバフローが発生するかどうかはタスクの処理にかかる時間にも依存します．

実際の処理ではiact_tskは割り込み処理1回につき，1回ずつ実行します．このiact_tskの処理結果に対してエラーを確認することになります．エラーが発生した場合は，起動要求

を再度発行するリトライか，処理できないとして処理を行わないかを判断する必要があります．

これを人の手で再現させるには，起動要求を出すタスク，割り込み処理などを動かす必要があります．たとえば，ボタン押下で割り込み処理が動くなど，外部との連携があり，この割り込みが発生するボタンを連打することなどで発生することもあります．しかし人の手で再現するにはばらつきがあったり，そんなに速く連打できないという問題があります．

ここでは疑似的にバッファオーバフローを発生させるように，3回連続で起動要求をかけてみました．実際の処理とは違うのですが，エラーを疑似的に発生させてみるとよくわかるでしょう．リスト 8をみてください．

〔リスト 8〕疑似的な複数回の要求発生（1回の場合）

```
/*
 *  割り込みハンドラ
 *  : 実際のシステムでは iact_tskは一つだけ呼びだすので
 */
void interrupt_handler(void)
{
    iact_tsk(TASK_ID);  /* タスクを起動(READY状態へ) */
}
```

〔リスト 9〕疑似的な複数回の要求発生（2回の場合）

```
/*
 *  割り込みハンドラ:
 *  疑似的に負荷を掛けてみたい場合，ハンドラを以下に入れ替える
 *  複数回であることを示しているだけで回数に意味はない.
 */
void interrupt_handler(void)
{
    error=iact_tsk(TASK_ID1);/* 入力タスクを起動(READY状態へ) */
    // エラー処理

    error=iact_tsk(TASK_ID1);/* 入力タスクを起動(READY状態へ) */
    // エラー処理
}
```

〔リスト 10〕疑似的な複数回の要求発生（3回の場合）

```
/*
 *  割り込みハンドラ:
 *  疑似的に負荷を掛けてみたい場合，ハンドラを以下に入れ替える
 *  複数回であることを示しているだけで回数に意味はない.
 */
sratic ER error;
void interrupt_handler(void)
{
    error=iact_tsk(TASK_ID);  /* タスクを起動(READY状態へ) */
       /* ログ出力error*/
    error=iact_tsk(TASK_ID);  /* タスクを起動(READY状態へ) */
       /* ログ出力error*/
    error=iact_tsk(TASK_ID);  /* タスクを起動(READY状態へ) */
       /* ログ出力error*/
}
```

〔リスト 11〕共通部分

```
/*
 *  起動されるタスク  タスク IDはTASK_ID
 */
void task(VP_INT exinf)
{
    while(1){
        message("処理をはじめます");
        proc_something();
        ext_tsk();          /* タスク終了(DORMANT状態へ) */
    }
}
```

これを実際に動かすと，iact_tskの三つ目でエラーが発生します．このエラーが発生したときは起動要求がキューイングされず，バッファオーバフローが発生します．

実際の割り込み処理でiact_tskが何らかの割り込みで1回ずつ発行される場合でも，タスクから見た場合，ほぼ同じ動きになります．1回目の要求でタスクに起動がかかり，2回目の要求で1回目のタスクが終了した後すぐに2回目のタスク起動がかかります．3回目の起動要求はバッファリングされず，エラーとして扱われ，タスクの起動は行われません．

このため，リスト 8のように起動されるタスクのログ出力はリスト 9，リスト 10と同じになります．

なお，起動されるタスクの共通部分はリスト 11のようになっているとします．

iact_tskに限らず，負荷が異常に高い状況などで，単位時間あたりいくつまで処理できるとするかは設計の範ちゅうですが，それでも，範囲を超えた場合のふるまいも盛り込んで設計しておく必要があります．

一つの例としては，範囲を超えた場合には何らかの通知を行い，リカバリの作業を行う方法があります．たとえば，通信などではエラーが発生した場合は，プロトコルを用意して再送するだけでなく，取りこぼしたことをダイアログで通知し，操作者にメッセージを見せて，リカバリ処理を行えるようにすることもあります．

また，高負荷時でのオーバフロー以外にも，思ったより低い基準でバッファオーバフローが発生していないかどうかを確認する必要があります．両方とも発生時のログを取っておき，「いつオーバフローが発生したのか？」，「それは本当に想定していた過負荷の基準を超えているのか？」などを確認できるようにしておくことも必要です．

なお，当たり前のことですが，これらのリカバリ処理や，ログ処理などは，事前に想定している負荷状態で正常にリカバリができること，同じくログが残せることを確認しておく必要があります．リカバリに失敗するリカバリ処理やログを残せないログ処理ほど情けないものはないでしょう．

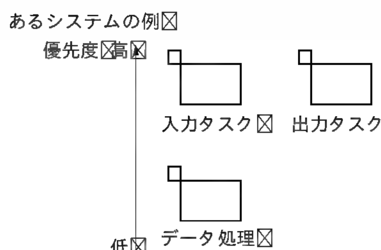
● タスクの優先順位

ここまでのところ，優先順位に関して説明しなかったのですが，タスクの起動要求の出し方はわかったので，最小構成の入力のタスク，出力のタスク，データ処理のタスクを図 8のような優先順位で動かしてみましょう．

外部からのリクエストで，データ処理とタスク管理を行うタスクから入力タスク，出力タスクを制御します．この構成では図 9のような動きになります．

外部からの入力を直接入力タスクに入れても良いが，タスクの初期化時間が要求する時間内に終わりそうにない場合などに利用され，システム全体の初期化処理などに向いています．

〔図8〕
データ処理タスクに起動要求を出す



また，この構成から，管理タスクを入力タスク，出力タスクより優先順位を高くすることもあります．この場合，タイムアウトは管理タスクの待ち時間で設定します．

では，ここで構成を変えて，割り込み処理から入力タスクを起動してみましょう．処理の流れは図10の太線で示したようになります．

ここまでの範囲では上記のように動かすことができますが，次のような場合に注意が必要です．このような場合，せっかくの速度差吸収がうまくいきません．

図11と図12の入力タスクの終了までの時間に注目してください．速度差を吸収するためには，入力タスクの終了までの時間を短くする必要があります．

タスクの強制終了

C言語API
ER ter_tsk(ID tskid);
パラメータ
ID tskid;　タスクID番号
リターンパラメータ
ER E_OK(正常終了)またはエラーコード
エラーコード
(E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_ILUSE, E_OBJ E_ILUSE：サービスコールの不正利用(自タスクを指定) E_OBJ　：オブジェクト状態エラー(対象タスクが休止状態)

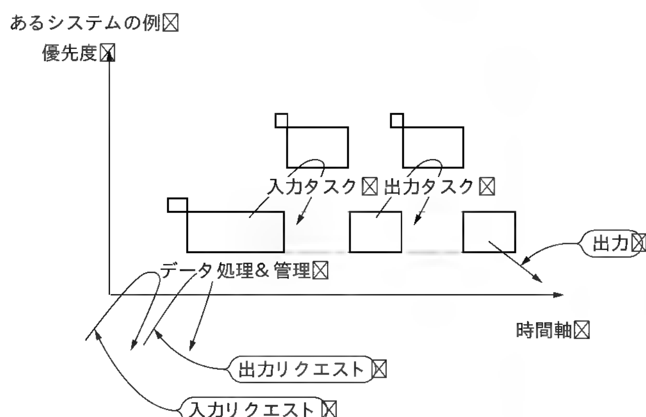
指定したタスクを強制的に休止状態にします．待ち状態，強制待ち状態にある場合は待ちを解除し，休止状態にします．

休止状態から再度起動するには，act_tskを使用しますが，このときタスク生成時の状態となります．

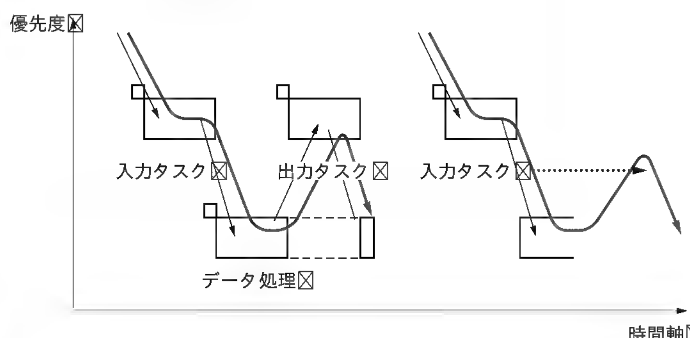
たとえば，タスク生成時のタスク優先度から変えていても戻されますし，使用途中のスタックも初期位置に戻されます．次のact_tskによるタスク起動では，過去に処理した結果に左右されません．つまり，サービスコールは過去の結果に左右されませんが，アプリケーションの作り方には左右されます．アプリケーションを過去の実行結果に依存しないように作成する必要があります．

過去の実行結果に左右されるとは，グローバル変数を初期化せずに使用したり，他のタスクに初期化してもらった変数をそのまま使うなどした場合です．動作する環境に依存していないかを考慮する必要があります．

〔図9〕要求をもらって，データ処理タスクが入力タスク，出力タスクを動かす



〔図10〕割り込み処理から入力タスクを動かし，データ処理タスク，出力タスクを動かす



複数回起動することもあり得るので，タスクを終了させる前に確保したメモリやセマフォなどの資源を解放しておく必要もあります．

開発初期にはact_tskを発行するのは一度だけのつもりが，機能拡張や設計の変更によって何回も発行することになったなどはよくある話なので，最初から考慮しておくことを勧めます．

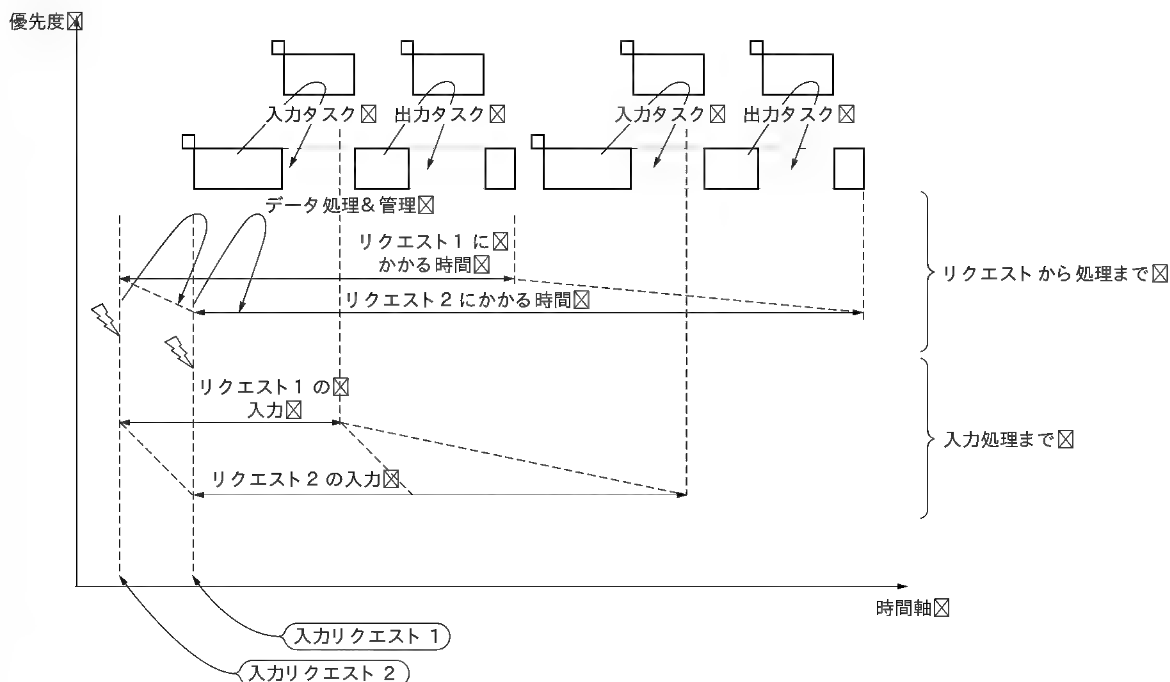
また，仕様上はsta_tskもact_tskの代わりに使用できそうですが，TOPPERS/JSPの実装はスタンダードプロファイルの範囲内なので，sta_tskは存在しません．

タスク優先度の変更

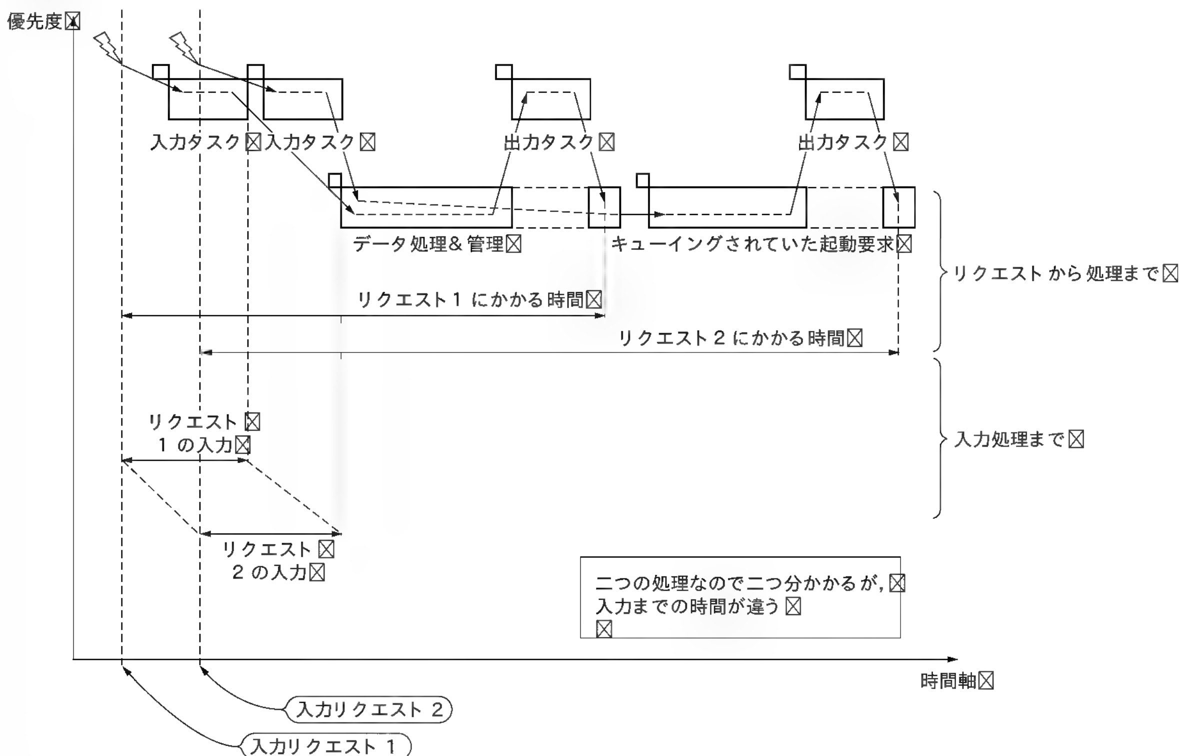
C言語API
ER chg_pri(ID tskid,PRI tskpri);
パラメータ
ID tskid;　タスクID番号 PRI tskpri　変更後のベース優先度
リターンパラメータ
ER E_OK(正常終了)またはエラーコード
エラーコード
(E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_ILUSE, E_OBJ

タスクの優先度を変更します．タスクの優先度は，ベース優

〔図11〕優先順位による処理順序の固定



〔図12〕優先順位による処理順序の固定（起動要求をキューイングした場合）

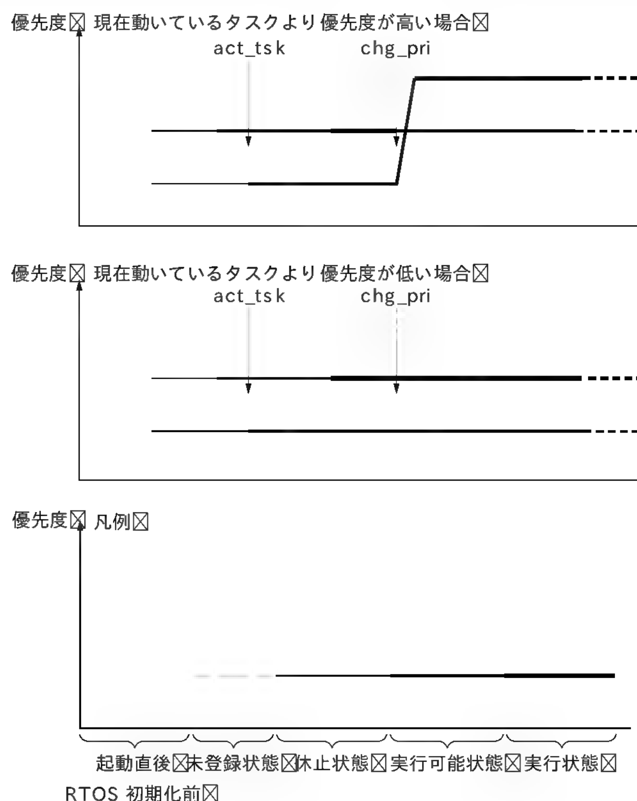


先度と現在優先度があります．TOPPERS/JSPではベース優先度と現在優先度とが同じ値になります．

ベース優先度は，タスクを生成した時点で決定される優先度です．chg_priでは，このベース優先度を変更します．変更したときに，chg_priを発行したタスクより高い優先度を指定した場合，タスクスイッチが発生します．変更したタスクの優先度が現在実行中のタスクより低い場合は何も起こりません．

ちなみに，現在優先度は一時的に優先度を変更して，排他制

〔図13〕タスクの優先度



御で発生する問題を解決するために使います．なお，TOPPERS/JSPではこの排他制御機構，ミューテックスのサポートはないので，ベース優先度＝現在優先度となります．

タスク優先度の参照

C言語API
ER get_pri(ID tskid,PRI *p_tskpri);
パラメータ
ID tskid;　タスクID番号
リターンパラメータ
ER E_OK(正常終了)またはエラーコード PRI *p_tskpri　対象タスクの現在優先度
エラーコード
(E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ

タスクの優先度を参照します．chg_priとは違い，現在の優先度を取得します．なお，TOPPERS/JSPではミューテックスのサポートはないので，ベース優先度＝現在優先度です．

● 再度，タスクの優先度について

優先度を変更するサービスコールが出てきたので，図13のように使ってみます．この例では，優先度を変えてから動作させるタスク群と，サービスコールを呼び出すタスクより優先度を実際に上げてみました(リスト12)．

〔リスト12〕優先順位：メインタスクの方が高い優先度

```
CRE_TSK(TASK_ID1, {TA_HLNG, (VP_INT)1,input_task,MID_PRIORITY,
                                                STACK_SIZE,NULL});
CRE_TSK(TASK_ID2, {TA_HLNG, (VP_INT)1,output_task,MID_PRIORITY,
                                                STACK_SIZE,NULL});

CRE_TSK(MAIN_TASK, {TA_HLNG|TA_ACT,0,main_task,MAIN_PRIORITY,
                                                STACK_SIZE,NULL});
```

タスク管理機能の総括

ここまででタスクを制御する最低限の機能を説明しました．これだけでは使えないという印象があるかもしれませんが，この部分はタスク管理の基礎になる部分です．

タスクの状態遷移が理解できれば，後のサービスコールの理解も早いのではないかと思います．キューイングやネストに関しては後で出てくるサービスコールでも使われている概念です．これがわかれば，後々の説明も取り付きやすくなるのではないでしょうか．

なお，メモリの使用制限が厳しい環境下で使うことを考慮した自動車制御用プロファイルなどでも，ここまでで説明した機能が利用できます．このプロファイルでは，制約タスクと呼ぶ，スタックを複数のタスクで共有する手法を利用できます．この制約タスク自体は，TOPPERS/JSPで提供していませんが，待ちに入るサービスコールを除いて，ほぼ同じ利用方法でタスク制御が行えます．

自動車制御用プロファイルが提供する機能は少ないのですが，使い方として制約があるため，プログラミングはより難しくなっています．ここでもタスクの制御は重要なサービスコールとして提供されています[注2]．

おわりに

ここまでで，タスクの管理機構について説明しました．ここで説明したサービスコールを使いこなせそうな感触をもっていただけたでしょうか？

次回は，引き続きタスクを直接操作する同期機構を説明します．これをきっかけにμITRON4.0仕様やTOPPERS Projectに興味をもっていただけたらと思っています．

きしだ・まさみ　(株)フルノシステムズ

注2：なぜ自動車制御用プロファイルでは待ちに入れないか？という点だが，タスクで使用するスタックを複数のタスクで共有する，制約タスクの機能を追加しているというのがその理由である．この制約タスク内ではスタックを共有することから，待ちに入るサービスコールを利用できない．

TMS320C6713搭載DSPスタータキットを使った

C++による DSP オブジェクト指向プログラミング

第1回 DSPスタータキットとアナログ信号入出力用クラス

◆三上 直樹

この連載では，C++言語によるDSPのプログラミングについて数回にわたって解説していきます．

VLIW(Very Long Instruction Word)アーキテクチャを採用した米国Texas Instruments社(以下，TI)のDSPであるTMS320C6711を搭載したDSPスタータキット(DSK)については拙著「C言語によるディジタル信号処理入門」[1]ですでに紹介していますが，このたび，その上位バージョンであるTMS320C6713を搭載したDSPスタータキットがTI社から発売されました．このDSKはTMS320C6711搭載のものとは異なり，2チャネルのA-D/D-A変換器をもつCODEC用LSIを搭載し，標本化周波数も最大で96kHzまで設定可能となっています．したがって，このDSKだけで従来よりも幅広い実験を行うことができるようになりました．

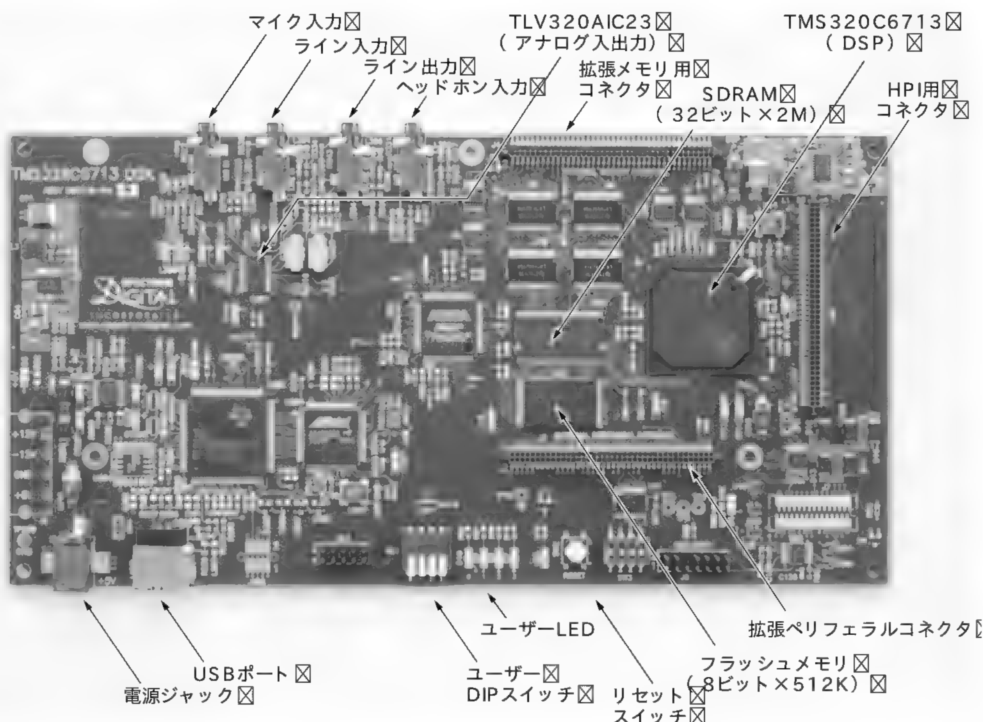
ところで，「C言語によるディジタル信号処理入門」では題名のとおり，C言語を用いてDSPのプログラムを作成しています．また，本誌2003年5月号では，特集記事として「オブジェクト指向の導入で開発効率向上！」という副題で，組み込み機器の開発手法を取り上げました．そこで，新しいDSKでのDSPのプログラム開発においても開発効率の向上を図るため，C++言語を使ってオブジェクト指向を取り入れたプログラミングに取り組んでみました．

1 TMS320C6713搭載DSPスタータキットの概要

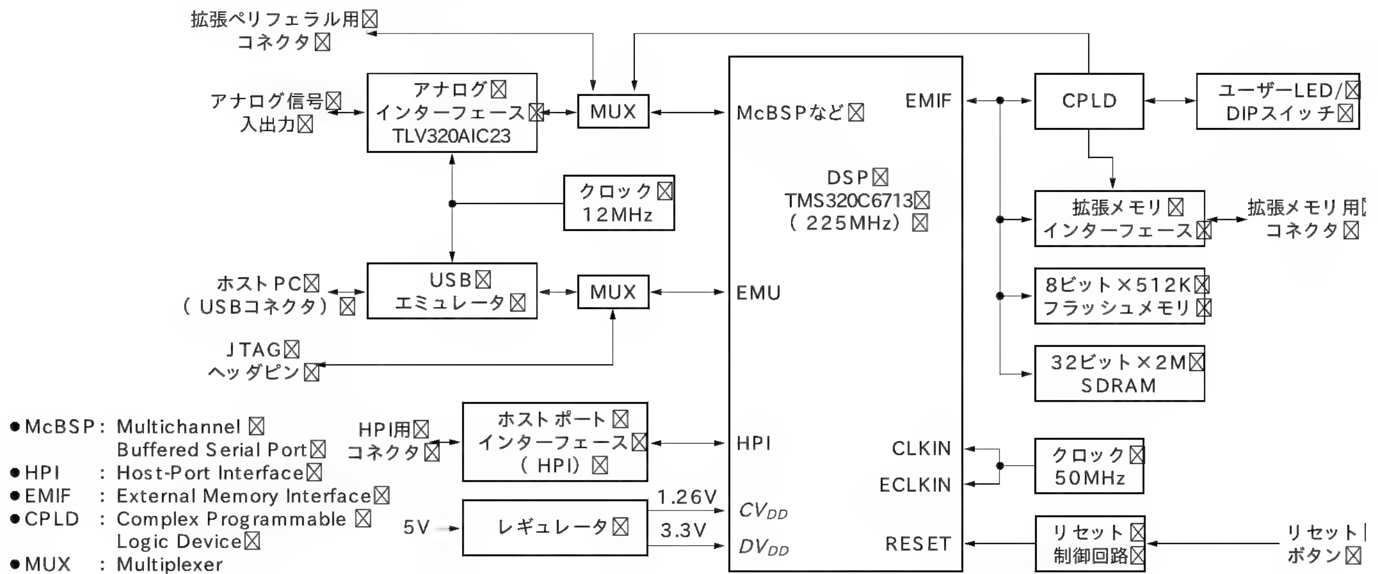
● DSKのハードウェア概要

TMS320C6713を搭載したDSPスタータキット[2](以下，C6713 DSK)のボードの外観を写真1に示します．TMS320C6711搭載のボード(以下，C6711 DSK)に比べて，若干大きくなっています．TMS320C6713についてはpp.134-136のコラム「TI社のDSP

〔写真1〕C6713 DSKボード



〔図1〕C6713 DSKボードの主要部のブロック図



ファミリとTMS320C6713の概要」で簡単に説明します．

図1にC6713 DSKボードの主要部のブロック図を，図2にメモリマップを示します．

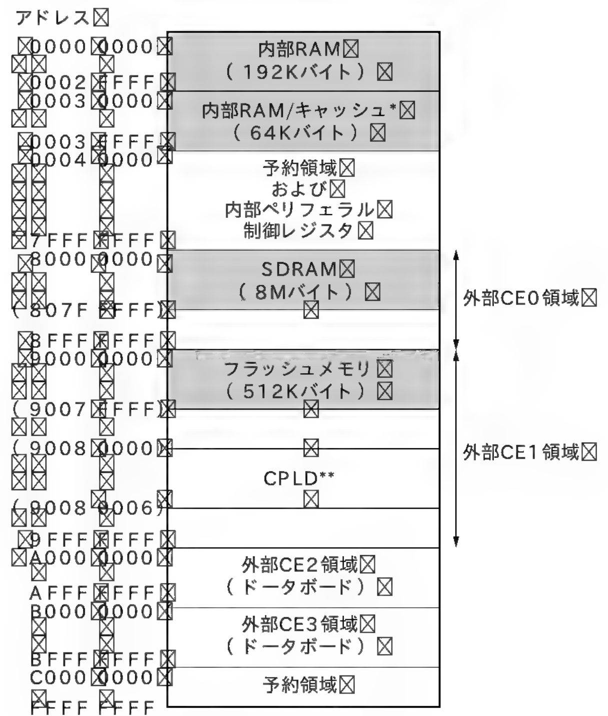
本ボードが従来のC6711 DSKボードと大きく異なる点は，プログラム開発用のPCとのインターフェースと，アナログ信号の入出力の部分です．

本ボードとプログラム開発用のPCは，USBポートで接続します．一方，従来のC6711 DSKではパラレルポートが使われていました．最近のノートPCはパラレルポート(IEEE1284準拠，25ピンD-Subコネクタ)をもたないものもありますが，USBが使えるので便利です．

アナログ信号の入出力のためには，24ビットのA-D変換器とD-A変換器を内蔵するCODEC用のTLV320AIC23[3]を搭載しています．このLSIは標本化周波数を変えることができ，最大で96kHzまで設定できます．また，A-D変換器とD-A変換器をともに2チャネル備えています．一方，従来のC6711 DSKは，アナログ信号の入出力は1チャネルのみで，標本化周波数が8kHzに固定されていました．

このTLV320AIC23は，TMS320C6713のMcBSP(Multichannel Buffered Serial Port)を介して接続されています．TMS320C6713は2チャネルのMcBSPをもっています．McBSP0はTLV320AIC23の制御用レジスタに，McBSP1はTLV320AIC23のデータにそれぞれ接続されています．なお，McBSPはDSKボードに搭載できるドータボード用コネクタ(拡張ペリフェラル用コネクタ)とも接続されています．そのため，McBSPとTLV320AIC23またはドータボード用コネクタの間には，マルチプレクサが入っており，CPLD(Complex Programmable Logic Device)の中に構成されているレジスタを書き換えることで，McBSPの接続先を決められるようになっています．

〔図2〕C6713 DSKのメモリマップ



* ：L2キャッシュまたは通常のメモリとして使用
**：ボード上のLED，ディップスイッチ用レジスタ，およびボードを制御するためのレジスタに割り当て
網かけされている部分は実装されているメモリを示す．

TMS320C6713は，外部との接続ポートとしてMcBSPのほかに，HPI(Host-Port Interface)をはじめとする複数のポートをもちます．しかし，McBSP以外のポートはドータボード用



TI社のDSPファミリとTMS320C6713の概要

TI社注Aは，1982年に16ビット固定小数点演算方式のTMS32010注Bを発表して以来，各種のDSPを発表しています．現在は同社からC2000，C5000，C6000の三つ注CのプラットホームのDSPが販売されています．各プラットホームはCPUコアの構成によりいくつかの製品群に分類されています．それらをまとめて表Aに示します．

次に，各プラットホームについて説明します．

● C2000

C2000プラットホームは，高性能なDSPコアをベースにして，モータ制御に必要な周辺機能を1チップに内蔵したDSPです．内蔵している周辺機能としてはA-D変換器，PWM（パルス幅変調）用回路，タイマ，シリアルインターフェースなどがあります．

▶C24x

C24シリーズのDSPは20MIPS（Million Instructions Per Second）のコアを持ち5Vで動作するシリーズと，40MIPSのコアを持ち，3.3Vで動作するシリーズがあります．

DSPコアはデータバスとプログラムバスをそれぞれ1組ずつ持つ，いわゆるハーバードアーキテクチャを採用しています．このコアはTMS320C25のコアに改良を加えたものになっています．

内蔵されているA-D変換器は10ビットで，変換速度が最速の製品では変換時間が375nsです．

内蔵ROMはフラッシュメモリのものと，マスクROMのものが提供されています．

▶C28x

C24xの上位互換のDSPで150MIPSのコアを持っています．DSPコアの動作電圧は1.8Vで，I/O部の動作電圧は3.3Vです．

このDSPコアは性能を上げるため，C24xのバス構造を拡張し，データバスがリード用とライト用に分離した構造になっています．

内蔵されているA-D変換器は12ビットで，変換速度が最速の製品では変換時間が80nsです．

内蔵ROMはフラッシュメモリのものが提供されており，現在マスクROMのものは開発中とのことです．

● C5000

C5000プラットホームは，携帯電話などのように限られた消費電力で高い処理能力が要求される機器のために開発されたDSPです．

このプラットホームには1チップ内にDSPコアとRISCコアを組み込んだシリーズもありますが，ここでは省略します．

▶C54x

C54xシリーズのDSPは，40MIPSから160MISPの処理能力をもつ多彩な製品が用意されています．なお，このシリーズでは1チップの中に最大で4個のDSPコアと共有メモリが集積されているものもあり，その処理能力は最大で532MIPSになります．

DSPコアのバス構造は，プログラムバスが1組，リード用のデータバスが2組，ライト用のデータバスが1組という構成になっています．

演算ユニットとして，ALU（Arithmetic Logic Unit）のほかにMAC（Multiply/Accumulate）をもち，ディジタル信号処理によく現れる積和の計算が高速に実行できます．また，誤り訂正符号を用いたシステムでの復号方式でよく用いられるビタビ（Viterbi）復号を効率よく実行できるようなユニットも備えています．

命令の語長は16ビットに固定されています．

▶C55x

C54xシリーズをさらに低電力化し，性能を約5倍に高めたDSPです．消費電力は400MHz動作時で，0.05mW/MIPSまで低消費電力化されています．

DSPコアのバス構造は，プログラムバスが1組C54xと変わりませんが，データバスはC54xのバス構造よりさらに拡張されています．リード用のデータバスが3組，ライト用のデータバスが2組という構成です．

また，ALUとMACもそれぞれ2組ずつ備えています．

〔表A〕現在販売されているTI社DSPプラットホーム

プラットホーム	プラットホームの特徴	DSPコア	DSPコアの特徴
C2000	モータ制御用の各種周辺機能を集積，フラッシュメモリ版とマスクメモリ版を提供	C24x	低価格で，最大で40MIPSの処理能力
		C28x	C24の上位互換で，最大で150MIPSの処理能力
C5000	低消費電力，高性能	C54x	低消費電力，高性能，最大で160MIPS†の処理能力
		C55x	C54の約5倍の処理能力を持ち，さらに低消費電力，最大で400MIPSの処理能力
C6000	改良VLIWアーキテクチャ，超高性能，キャッシュを内蔵	C62x	改良されたVLIWアーキテクチャを持つ固定小数点演算DSP
		C64x	C62xの上位互換で，その1桁高い処理能力を持つ固定小数点演算DSP
		C67x	浮動小数点演算器を備える以外はC62xと同じアーキテクチャ

†：1チップ当たりDSPコアが一つの製品の値．1チップ内にDSPコアを四つもつ製品では最大で532MIPS．

注A：日本テキサス・インスツルメンツのDSPに関するWebサイト http://www.tij.co.jp/jsc/docs/dsps/index.htm からたどっていくと，TI社のDSPやDSKを含む開発ツールの情報を得ることができる．
注B：最初の製品はNMOSで，処理能力は5MIPSだった．
注C：そのほか，OMAPというプラットホームがあるが，これはDSPコアとARMコアを1チップにしたものなので，ここでは省略する．

命令の語長は8ビットから48ビットの可変語長になっています．

● C6000

C6000プラットホームは並列処理の一種であるVLIW（Very Long Instruction Word）方式のアーキテクチャを採用した超高性能のDSPです．C6000のVLIWアーキテクチャ[注D]は従来のものを改良してメモリの利用効率を向上したもので，C62x，C67xではVelociTI，C64xではVelociTI.2と呼ぶアーキテクチャを採用しています．

DSPコアは大きく二つのブロックに別れ，それぞれのブロックにL，S，M，Dという機能の異なる四つの実行ユニットをもっています．これらの実行ユニットは，最大で八つが並行して処理を行います．

このプラットホームのDSPは，外部に高速でないメモリを使用した場合でも高い処理能力を実現するため，内蔵のRAMをキャッシュメモリとして使うことができます．キャシュメモリの構成には2通りあり，1レベルのものと2レベルのものがあります．

2レベルのキャッシュ構成のDSPでは，1次キャッシュがキャッシュ専用でプログラム用とデータ用に分かれています．2次キャッシュはプログラムとデータが共通になっており，通常のメモリとの共用になっています．

このプラットホームのDSPは，プログラム開発をC言語で行うことを前提として設計されています．

最初に出た製品は，クロックの最大値が200MHzで，このときの処理能力は1600MIPSでした．新しいものでは，クロックの最大値が720MHzで，このときの処理能力は5760MIPSとなっています．

▶C62x

改良されたVLIWアーキテクチャ（VelociTI）を持つ最初の固定小数点演算方式DSPです．

命令体系は，RISCプロセッサと似た単純な命令になっており，またいわゆるロードストアアーキテクチャに基づく命令体系になっています．つまり，演算はレジスタ同士で行い，メモリにアクセスするのはロード命令とストア命令だけに限定されています．

もっとも高速のものはクロックの最大値が300MHzで，そのときの処理能力は2400MIPSです．

なお，C620xのシリーズは1レベルのキャッシュを，C621xのシリーズは2レベルのキャッシュを内蔵しています．

▶C67x

C62xシリーズに対して，浮動小数点演算用のハードウェアをオンチップに追加した製品です．この点を除いてはC62xシリーズとの互換性をもちます．命令体系も，浮動小数点命令を除くと，C62xとまったく同じです．したがって，研究開発段階ではC67xで行い，製品として大量生産を行う場合にはC67xより価格が安いC62xを使うという使い分けが可能です．

もっとも高速のものはこの連載で取り上げているDSKに搭載されているTMS320C6713です．そのクロックの最大値は225MHzで，そのとき，浮動小数点演算の性能で1350MFLOPS（Mega Floating-Point Operations Per Second）の処理能力があります．

なお，C670xのシリーズは1レベルのキャッシュを，C671xのシリーズは2レベルのキャッシュを内蔵しています．

▶C64x

C62xの約10倍の処理能力をもちながら，消費電力を約1/3に抑えた固定小数点演算方式のDSPです．

C64xはC62x，C67xのVelociTIアーキテクチャをさらに改良したVelociTI.2アーキテクチャを採用し，メモリの使用効率をさらに向上させています．また，このシリーズのDSPはすべて2レベルのキャッシュを内蔵しています．

命令体系は，C62xのものに加えて，ディジタル通信，静止画像および動画像処理，グラフィクスなどのアプリケーションを効率良く実行するために，ガロア体乗算などの特殊な命令が追加されています．また，加算，乗算，差分絶対値計算などでは四つの演算を一つの実行ユニットで同時に実行できるようにして命令の並列度を高めています．

もっとも性能が高いものは，クロックの最大値が720MHzで，そのときの処理能力は5760MIPSです．

● TMS320C6713の概要

この連載で取り上げるDSKに搭載されているTMS320C6713は，従来のDSKに搭載されていたTMS320C6711の上位互換DSPです．このブロック図を図A（p.136）に示します．図Aで，網かけされている部分は，新たに追加されたものまたは機能が拡張されたものを示します．また，表B（p.136）に二つのDSPの比較を示します．

TMS320C6713のクロック最大値は225MHzで，最大処理能力は1800MIPS，最大浮動小数点演算能力は1350MFLOPSになりました．内蔵RAMも拡張され，192Kバイトが新たに追加されました．この内蔵RAMは，最大で64Kバイトを2次キャッシュとして使うことができます．内蔵ペリフェラルは従来のもののほかに外部インターフェース用としてMcASP（Multichannel Audio Serial Port），I^2C（Inter-Integrated Circuit）バス，GPIO（General-Purpose Input/Output）ポートが内蔵されました．

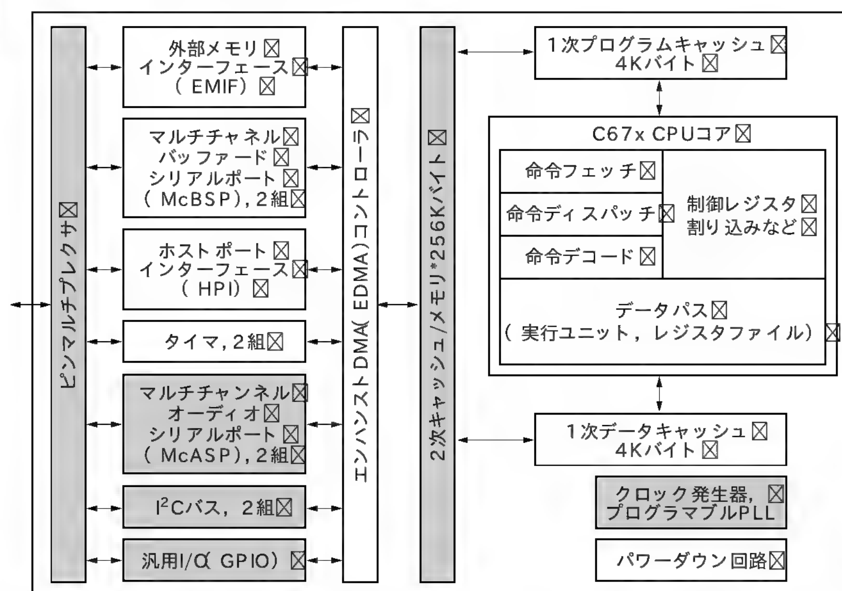
このように内蔵ペリフェラルが増えたので，TMS320C6713では，内蔵ペリフェラルに関する信号はすべてが外部端子に引き出されているというわけではありません．その代わり外部端子と内蔵ペリフェラルの間にマルチプレクサを入れ，必要に応じてプログラムで切り替えて使うようになっています．ただし，デフォルトの状態ではTMS320C6711が備えていた内蔵ペリフェラルが外部端子に接続されています．

また，TMS320C6713では新たに，プログラマブルPLLを内蔵し，クロックの倍率や分周比をプログラムで設定できるようになりました．これによりDSKの外部からのクロックが1系統ですむようになったため，外部クロック系統のハードウェアを簡略化することができます．

注D：詳しくは，本誌1999年1月号「VLIWアーキテクチャを採用した新世代のDSP（前編）」を参照．

Column 1
（つづき）

〔図A〕TMS320C6713のブロック図



網かけ部分がTMSC320C6713で新たに追加または拡張された部分．

*2次キャッシュ/メモリは最大で64Kバイトを2次キャッシュとして使用可能．残りは通常のプログラムメモリまたはデータメモリとして使用．

〔表B〕TMS320C6711とTMS320C6713のおもな性能の比較

	TMS320C6711GFN	TMS320C6713GDP
最小マシンサイクル	6.67ns（クロック周波数：150MHz）	4.44ns（クロック周波数：225MHz）
最大処理能力	1200MIPS，8命令/サイクル	1800MIPS，8命令/サイクル
最大演算能力	900MFLOPS，6演算/サイクル	1350MFLOPS，6演算/サイクル
データ形式	8/16/32ビット，固定小数点	←
	32/64ビット，浮動小数点	←
命令形式	32ビット×8命令	←
アドレス空間	4Gバイト（アドレスはバイト単位）	←
内蔵メモリ	命令用1次キャッシュ　4Kバイト	←
	データ用1次キャッシュ　4Kバイト	←
	2次キャッシュ/RAM　64Kバイト	←
	無	RAM専用 192Kバイト
汎用レジスタ	32ビット×16個×2組	←
実行ユニット	8	←
内蔵ペリフェラル	EMIF，HPI，Timer，McBSP	←
	無	McASP，GPIO，I2C
プログラマブルPLL	無	×4～25および÷1～32
製造プロセス技術	0.18μm，CMOS	0.13μm，CMOS
電源電圧	1.8V（内部），3.3V（入出力）	1.26V（内部），3.3V（入出力）
パッケージ	256ピン BGA	272ピン BGA

値などはDSKに搭載されているDSPのもの，「←」は左に同じという意味．

コネクタに接続されているだけで，C6713 DSKではとくに使われているわけではありません．

ドータボード用コネクタとして，拡張ペリフェラル用のほかに，拡張メモリ用とHPI用のコネクタが実装されています．

そのほか，ボードをリセットするためのプッシュスイッチや，ユーザーが自由に使うことのできるDIPスイッチとLED，ブート方法などを切り替えるためのスイッチがそれぞれ搭載されています．

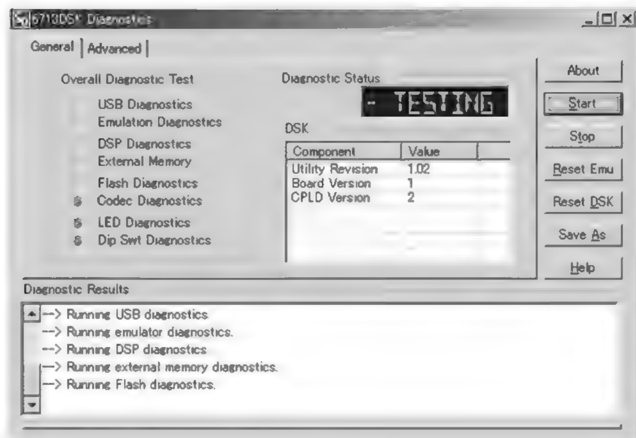
JTAGヘッダピンは，TI社XDS560エミュレータなどを接続してプログラム開発するときに利用するためのものです．

● DSKのソフトウェアの概要

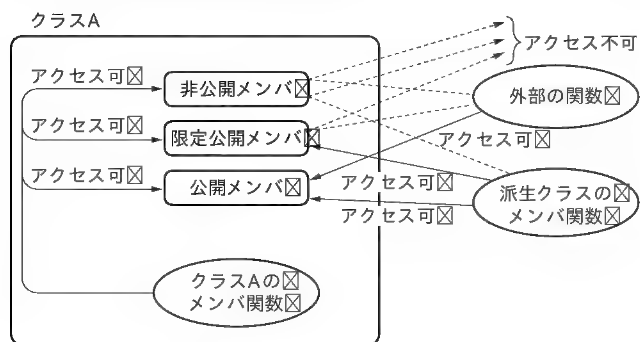
C6713 DSKには，プログラム開発のための統合環境であるCode Composer Studio（以下，CCS）が付属しています．このCCSには，コード生成ツール（C/C++コンパイラ，アセンブリオプティマイザ，リンカなど）や，デバッガなどが含まれています．筆者のDSKに付属しているCCSのバージョンは2.20で，Windows 98SE/2000/XPを搭載したPCに対応しています．

なお，DSKの動作確認のためのユーティリティは，CCS2.1

〔図3〕DSKの動作診断用ユーティリティ実行のようす



〔図4〕メンバに対するクラスのアクセスレベル



までのキャラクタベースのものからGUIベースのものに変更されています．このユーティリティを実行している最中のようすを図3に示します．

2 クラスのメンバへのアクセスと継承

オブジェクト指向プログラミングの中で重要な概念の一つがカプセル化(encapsulation)です．C++では，これを実現する言語要素がクラス(class)ということになります．クラスとは，データと関数を一つのパッケージにしたものです．クラスに含まれる要素をメンバ(member)と呼びます．メンバの中でもさらに，データはデータメンバ(data member)，関数はメンバ関数(member function)[注1]と呼びます．

クラスを作る際に，すでに存在するクラスの機能を拡張する形で新たなクラスを作成することができます．これを継承

〔表1〕継承とアクセスレベルの関係

基底クラスでのアクセス権限	継承時のアクセス指定子	派生クラスでのアクセス権限	派生クラスのユーザーからのアクセス
private	private	アクセス不可	アクセス不可
	protected		
	public		
protected	private	private	
	protected	protected	
	public		
public	private	private	
	protected	protected	
	public	public	アクセス可

(inheritance)といいます．これもオブジェクト指向プログラミングの中で重要な概念の一つです．継承により新たに作成したクラスを派生クラス(derived class)[注2]と，元のクラスを基底クラス(base class)[注3]といいます．

DSKボードの初期化やアナログ信号入出力は，クラスを使って実現します．そこで，以下ではクラスのメンバへのアクセスレベルと，継承の種類によって基底クラスのメンバのアクセスレベルがどうなるかについて簡単にまとめておきます．

オブジェクト指向プログラミングを行ううえで重要な要素の一つに，情報の隠ぺいがあります．これを実現するために，C++ではクラスのメンバに対して三つのアクセスレベルを規定しています．それは，公開(public)，限定公開(protected)[注4]，非公開(private)の三つです．C++のキーワードとしては，「public:」，「protected:」，「private:」を使います．この三つのレベルについてまとめたものを図4に示します[注5]．

図4からわかるように，クラスのメンバ関数からは，そのクラス内のどのメンバもアクセスが可能です．クラスの外部関数からは，公開メンバのみアクセスが可能です．派生クラスのメンバ関数からは，派生の元になったクラス(基底クラス)の公開メンバはもちろんのこと，限定公開メンバへのアクセスも可能です．したがって，限定公開は派生クラスを作ったときにはじめて意味をもちます．

次に，基底クラスのメンバのアクセスレベルが3種類の継承方法，つまり公開継承，限定公開継承，非公開継承により，派生クラスではどのアクセスレベルになるのか，および派生クラスのユーザーからは基底クラスのどのメンバにアクセスできるのかということを表1にまとめて示します．

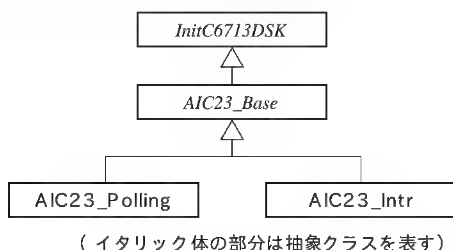
通常よく使うのが公開継承で，これはis-a関係[注6]を実現す

注1：データメンバを単にメンバと呼ぶこともある．また，メンバ関数をメソッド(method)と呼ぶこともある．本連載では，「データメンバ」，「メンバ関数」という用語を使うことにして，メンバという用語はデータメンバとメンバ関数を総称して呼ぶ場合に使う．
注2：導出クラスと呼ぶこともある．
注3：基本クラスと呼ぶこともある．
注4：被保護という用語を使う場合もある．
注5：フレンド(friend)関数からのアクセスについては省略する．これについては，フレンド関数を使うときに説明する．
注6：派生クラスは基底クラスの一種であるという関係．

るために使います．公開継承で作成された派生クラスのユーザーからは，基底クラスのアクセスレベルは変化しません．したがって，公開派生クラスのユーザーからは基底クラスの公開メンバのみアクセス可能です．一方，非公開継承および限定公開継承は限られた特別の場合にのみ使われ，is-a関係を実現するためではなく，「それを実装手段とする」という関係を実現する場合に使います[注7]．非公開および限定公開継承で生成されるクラスのユーザーからは，基底クラスのどのメンバもアクセスすることができません．

なお，非公開継承および限定公開継承で生成される派生クラスからは，基底クラスの限定公開および公開メンバがアクセスできます．したがって，派生クラスのメンバ関数から基底クラスのメンバへのアクセスは，継承の種類には関係ありません．これはすでに図4で示しているとおりです．

〔図5〕作成するクラスの継承のようす



（イタリック体の部分は抽象クラスを表す）

3 DSKボードの初期化とアナログ信号入出力用クラスの作成

最初に，DSKボード初期化のための抽象クラス[注8] `InitC6713DSK`を作成します．次に，これを基底クラスとして，アナログ信号入出力のための限定公開継承による派生クラス`AIC23_Base`を作成します．クラス`AIC23_Base`も抽象クラスとします．さらに，このクラスの公開派生クラスとして，アナログ信号入力のためにポーリング方式を使うクラス`AIC23_Polling`と，割り込みを用いてアナログ信号入力を行うクラス`AIC23_Intr`を作成します．

これらのクラスの継承関係を図5[注9]に示します．四つのクラス中，`InitC6713DSK`と`AIC23_Base`は抽象クラスです．したがってユーザーが直接インスタンス化できるクラスは，`AIC23_Polling`と`AIC23_Intr`の二つだけです．

● DSKボード初期化のためのクラス（**InitC6713DSK**）

DSKボードの初期化としては，DSPに関する二つの設定が必要になります．この中でとくに重要なのはDSPのクロックに関係するPLLの設定です．TMS320C6713の場合，内部のクロックや外部メモリのためのクロックは，外部から入力されたクロックを元にして，DSPに内蔵するPLLで生成しています．このPLLはプログラマブルになっているので，適切な値に設定する必要があります．もう一つは2次キャッシュの

〔リスト1〕C6713 DSKボード初期化のためのクラス

```
// ---------------------------------------------
// C6713 DSKボードの初期設定
//  作成者:  三上直樹, 2003/08/02
// ---------------------------------------------
#ifndef MK_InitC6713DSK

#define CHIP_6713 1
#include <dsk6713.h>
#include <csl.h>
#include <csl_cache.h>

// C6713 DSKを使うための基底クラス
class InitC6713DSK
{
public:
// 2次キャッシュのサイズを設定する定数を列挙型で定義
    enum L2mode { cache0K, cache16K, cache32K, cache48K,
                  cache64K = 7 };
protected:
    InitC6713DSK(const L2mode banks = cache64K);
    virtual ~InitC6713DSK() = 0;
};

#define MK_InitC6713DSK
#endif
```

（注釈：`#define CHIP_6713 1` → TMS320C6713を使う場合は，この定義が必要／`InitC6713DSK(const L2mode banks = cache64K);` → コンストラクタの宣言／`virtual ~InitC6713DSK() = 0;` → 純粋仮想デストラクタの宣言）

（a）InitC6713DSK.hpp

```
// ---------------------------------------------
// C6713 DSKボードの初期設定
//  作成者:  三上直樹, 2003/08/05
// ---------------------------------------------
#include "InitC6713DSK.hpp"

// コンストラクタの定義
InitC6713DSK::InitC6713DSK(const L2mode banks)
{
    CSL_init();
    IRQ_resetAll();
    CACHE_setL2Mode((CACHE_L2Mode)banks);
    CACHE_enableCaching(CACHE_CE00);

    DSK6713_init();
}

// 純粋仮想デストラクタの定義
InitC6713DSK::~InitC6713DSK() {}
```

（注釈：`CSL_init();` → チップサポートライブラリCSLのための初期化／`IRQ_resetAll();` → GIE，IER，IFRのリセット／`CACHE_setL2Mode(...)` → 2次キャッシュのサイズ設定／`CACHE_enableCaching(CACHE_CE00);` → 0x80000000～0x80FFFFFFを2次キャッシュの対象領域に設定／`DSK6713_init();` → ボードサポートライブラリ（BSL）によるDSPの初期化）

（b）InitC6713DSK.cpp

注7：派生クラスが基底クラスを使って実装するという関係．

注8：抽象クラスについては，p.139のコラム「抽象基底クラスと仮想関数」を参照．

注9：図5は，UML（Unified Modeling Language）のクラス図の描きかたに従っている．クラス図では，抽象クラスの名前をイタリック体で書くことになっている．

設定です．

リスト1(a)にDSKボード初期化のためのクラスInitC6713DSKに関するヘッダファイル(InitC6713DSK.hpp)を，リスト1(b)に定義の本体(InitC6713DSK.cpp)を示します．

(1)リスト1(a)，ヘッダについて

DSPの設定に関わる部分については，CCSに付属する形で提供されているCSL(Chip Support Library)[4]のAPIを使います．そのためのインクルードファイルがcsl.hです．これを使う場合，ソースプログラムの中で事前に使用するDSPの型名に応じてシンボルを定義しておく必要があります[注10]．TMS320C6713では以下のように定義します．

```
#define CHIP_6713 1
```

もう一つのインクルードファイルcsl_cache.hはキャッシュの設定に関するヘッダファイルです．

クラスInitC6713DSKのコンストラクタは，公開メンバではなく限定公開メンバとして宣言しています．これは，このクラスを継承した派生クラス以外からはインスタンス化をできないようにするためです．

▶ 公開メンバ

公開メンバとして，DSPの2次キャッシュ(L2キャッシュ)のサイズを設定するためのシンボルがL2modeというタグ(Tag)名で列挙型(enum)として宣言されています．このように，シンボルをクラスの内部で定義すると，シンボルのグローバル化が避けられるため名前の衝突防止に役立ちます．

▶ 限定公開メンバ

限定公開メンバとして，コンストラクタおよび純粋仮想デストラクタが宣言されています．

コンストラクタInitC6713DSK()の引き数には，デフォルト引き数値cache64Kを指定しており，引き数を指定しない場合は2次キャッシュのサイズを最大の64Kバイトに設定するようにしています．

最後に宣言されているデストラクタは純粋仮想デストラクタです．この宣言は，クラスInitC6713DSKを抽象基底クラス(abstract base class)[注11]にするために行いました．そのため，クラスInitC6713DSKを使う場合は，その派生クラスを作る必要があります．

(2)リスト1(b)，定義本体について

▶ **Init6713DSK()**

コンストラクタInit6713DSK()の中で使われている四つの関数，CLS_init()，IRQ_resetAll()，CACHE_setL2Mode()，CACHE_enableCaching()はCLSのAPIと

Column 2　抽象基底クラスと仮想関数

抽象クラス(abstract class)とはインスタンス(instance)を作成できないクラスのことです．インスタンスとは，実際にメモリの上に置かれて使えるようになったクラスの実体のことで，オブジェクト(object)と呼ぶこともあります．なお，抽象クラス以外のクラスはインスタンスを生成でき，これを具象クラス(concrete class)と呼びます．

たとえば，AbstractClassという名前の抽象クラスが宣言されているものとすると，次のようになります．

```
AbstractClass x;  ← コンパイルエラー
AbstractClass *ptr_x;  ← OK
ptr_x = new AbstractClass;  ← コンパイルエラー
```

この例で，ポインタの場合にはOKなのは，まだクラスの実体ができていないためです．しかし，new演算子を使って，このポインタに対応する実体を作成しようとすると，コンパイラがエラーを出します．

それではどのようなクラスが抽象クラスかというと，純粋仮想関数(pure virtual function)[注E]を持ったクラスが抽象クラスです．クラスのメンバ関数の頭に「virtual」を付けると，その関数は仮想関数となります．さらに，仮想関数の宣言の際に，右側に「= 0」を付けると純粋仮想関数となります．これらの例を次に示します．

```
virtual void func1();  ← 仮想関数
virtual void func2() = 0;  ← 純粋仮想関数
```

あるクラスに対して，そのデータメンバやメンバ関数を継承し，さらに機能を拡張することを派生といいます．そのとき元になったクラスを基底クラス，拡張されたクラスを派生クラスといいます．仮想関数とは，基底クラスのメンバ関数であって派生クラスから再定義できるような関数です．この再定義はオーバライド(overriding)とも呼びます．純粋仮想関数とは，派生クラスから再定義されることを前提とする関数で，基底クラスでは単に宣言するだけのものです．

抽象クラスは必然的に派生クラスの基底クラスとなるので，抽象基底クラスでもあるわけです．

注E：仮想関数は，同じ関数でもクラスが違えば別の動作を行うという，オブジェクト指向の重要な概念の一つである多態性(ポリモーフィズム；polymorphism)を具体化する手段を提供する．

注10：この定義は，プロジェクトのコンパイラに関するオプションの設定でも行うことができる．

注11：抽象基底クラスについては，コラム「抽象基底クラスと仮想関数」を参照．

〔リスト 2〕アナログ信号入出力のための基底クラス

```
//--------------------------------------------------------------
// C6713 DSKボード搭載のTLV320AIC23でアナログ信号入出力を行うための基底クラス
//  作成者:  三上直樹, 2003/08/04
//--------------------------------------------------------------
#ifndef MK_AIC23_Base

#include "InitC6713DSK.hpp" ←── DSKボード初期化用⊠
#include <csl_mcbsp.h>

// AIC23_Baseクラスの宣言, InitC6713DSKクラスを限定公開継承
class AIC23_Base : protected InitC6713DSK
{
public:
// 標本化周波数を設定するための定数を列挙型で定義
    enum AICfs { fs96kHz = 0x001D, fs48kHz = 0x0001,
                 fs44kHz = 0x0023,fs32kHz = 0x0019,
                 fs24kHz = 0x0041, fs16kHz = 0x0059,
                 fs08kHz = 0x000D };
    AIC23_Base(const AICfs fs_set = fs48kHz,
               const L2mode banks = cache64K);
    void RegSet(const unsigned short reg,
                const unsigned short val);
    void Read(short &chL, short &chR);
    void Read(float &chL, float &chR);
    void Read(short ch[]);  ←── ch[0]: 左チャネル⊠
    void Read(float ch[]);       ch[1]: 右チャネル⊠
    void Write(const short chL, const short chR);
    void Write(const float chL, const float chR);
    void Write(const short ch[]);  ←── ch[0]: 左チャネル⊠
    void Write(const float ch[]);       ch[1]: 右チャネル⊠
protected:
// McBSP1のレジスタのアドレスを列挙型で定義
    enum { DXR = 0x01900004, DRR =
       0x01900000, SPCR = 0x01900008 };
private: ←── McBSP0: TLV320AIC23の⊠コントロールレジスタ⊠
    MCBSP_Handle hControl;
    MCBSP_Handle hData;
    const unsigned short fs; ←
    union { unsigned int buf; short chn[2]; } xn, yn;  ←── McBSP1: ⊠TLV320⊠AIC23の⊠データ⊠
    void Config();
    virtual unsigned int AIC23_Read() const = 0;
// McBSP1による, MTLV320AIC23への32ビットのデータの書き込み
    void AIC23_Write(const unsigned int data)
                             { *( unsigned int *)DXR = data; };
};

// TLV320AIC23からのデータの読み込み(データの型: short)
inline void AIC23_Base::Read(short &chL, short &chR)
{
    xn.buf = AIC23_Read();
    chL = xn.chn[1];
    chR = xn.chn[0];
}

// TLV320AIC23からのデータの読み込みと正規化(データの型: float)

inline void AIC23_Base::Read(float &chL, float &chR)
{
    xn.buf = AIC23_Read();
    chL = ((float)xn.chn[1])*3.051758e-5f;
    chR = ((float)xn.chn[0])*3.051758e-5f;
}

// TLV320AIC23からのデータの読み込み(データの型: shortの配列)
inline void AIC23_Base::Read(short ch[])
{
    xn.buf = AIC23_Read();
    for (int k=0; k<2; k++) ch[k] = xn.chn[1-k];
}

// TLV320AIC23からのデータの読み込みと正規化(データの型: floatの配列)

inline void AIC23_Base::Read(float ch[])
{
    xn.buf = AIC23_Read();
    for (int k=0; k<2; k++)
        ch[k] = ((float)xn.chn[1-k])*3.051758e-5f;
}

// TLV320AIC23へのデータの書き込み(データの型: short)
inline void AIC23_Base::Write(const short chL, const short chR)
{
    yn.chn[1] = chL;
    yn.chn[0] = chR;
    AIC23_Write(yn.buf);
}

// 正規化されたデータのTLV320AIC23への書き込み(データの型: float)

inline void AIC23_Base::Write(const float chL, const float chR)
{
    yn.chn[1] = (short)(chL*32768.0f);
    yn.chn[0] = (short)(chR*32768.0f);
    AIC23_Write(yn.buf);
}

// TLV320AIC23へのデータの書き込み(データの型: shortの配列)
inline void AIC23_Base::Write(const short ch[])
{
    for (int k=0; k<2; k++) yn.chn[1-k] = ch[k];
    AIC23_Write(yn.buf);
}

// 正規化されたデータのTLV320AIC23への書き込み(データの型: floatの配列)

inline void AIC23_Base::Write(const float ch[])
{
    for (int k=0; k<2; k++)
        yn.chn[1-k] = (short)(ch[k]*32768.0f);
    AIC23_Write(yn.buf);
}

#define MK_AIC23_Base
#endif
```

(a) AIC23_Base.hpp

して提供されている関数です．CSLのAPIを使う場合[注12]は，それを呼び出す前に必ず関数`CSL_init()`を実行する必要があります[4]．

`IRQ_resetAll()`は割り込みに関するレジスタをすべてクリアし，割り込みを禁止します．

その次の二つの関数は，2次キャッシュに関する設定を行います．`CACHE_setL2Mode()`はコンストラクタの引き数で指定される2次キャッシュのサイズを設定します．`CACHE_enableCaching()`は外部メモリの領域を選択的に2次キャッシュの対象となるように設定します．このリストの設定では，DSKに搭載されている外部メモリの領域をすべて含むアドレス0x8000 0000～0x80FF FFFFの領域が2次キャッシュの対象となります．

最後の`DSK6713_init()`は，BSL(Board Support Library)[注13]としてDSKに付属の形でサポートされているAPI関数です．この関数の中では，外部メモリのインターフェースに使われる

注12：CSLのAPIは，CCSが提供するDSP/BIOSカーネルの下で使うということが前提になっている．しかし，DSP/BIOSを使う場合，設定が煩雑なのでこの連載ではDSP/BIOSを使用しない．その場合には，関数`CSL_init()`を事前に実行する必要があるということがユーザーズガイド[4]に載っている．

注13：`ti¥c6000¥dsk6713¥lib`の`dsk6713bsl.zip`を展開すると，BSLのソースリストを見ることができる．なお，BSLのユーザーガイドは現在のところC6711 DSK対応のもの(文献番号SPRU432A)しか準備されていないようである．

〔リスト2〕アナログ信号入出力のための基底クラス(つづき)

```
//-------------------------------------------------------------
// C6713 DSKボード搭載のTLV320AIC23でアナログ信号入出力を行うための基底クラス
//  作成者: 三上直樹, 2003/08/21
//-------------------------------------------------------------
#include "AIC23_Base.hpp"

 // コンストラクタの定義
AIC23_Base::AIC23_Base(const AICfs fs_set, const L2mode banks)
                       : InitC6713DSK(banks), fs(fs_set)
{
// McBSP0の設定のための構造体(データの送信は16ビット単位)
    MCBSP_Config mcbsp0 = { 0x00001000, 0x00000000, 0x00000040,
                            0x20001363, 0x00000000, 0x00000000,
                            0x00000000, 0x00000A0A };
// McBSP1の設定のための構造体(データの送受信は32ビット単位)
    MCBSP_Config mcbsp1 = { 0x00000000, 0x000000A0, 0x000000A0,
                            0x20000001, 0x00000000, 0x00000000,
                            0x00000000, 0x00000003 };
// McBSP0, McBSP1のオープン
    hControl = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
    hData = MCBSP_open(MCBSP_DEV1, MCBSP_OPEN_RESET);

    MCBSP_config(hControl, &mcbsp0);   ← McBSP0の設定
    MCBSP_config(hData, &mcbsp1);      ← McBSP1の設定
// McBSP0によるTLV320AIC23のレジスタの設定開始
    MCBSP_start(hControl, MCBSP_XMIT_START |
                MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 100);

    RegSet(0x0F, 0);   ← TLV320AIC23のリセット
    Config();          ← TLV320AIC23のレジスタの設定
    MCBSP_read(hData);   ← ダミー読み込み
// McBSP0の動作の開始
    MCBSP_start(hData, MCBSP_XMIT_START | MCBSP_RCV_START |
                MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);

    MCBSP_read(hData);   ← ダミー読み込み
}

// TLV320AIC23の指定されたレジスタの設定
void AIC23_Base::RegSet(const unsigned short reg,
                        const unsigned short val)
{
    while (!MCBSP_xrdy(hControl));
    MCBSP_write(hControl, (reg << 9) | (val & 0x1FF));
}

// TLV320AIC23のレジスタ(R0～R9)の設定
// ライン入力のゲイン: +6 dB
// A-D/D-Aのビット長:    16ビット
void AIC23_Base::Config()
{
    unsigned short conf[10] = { 0x001B, 0x001B, 0x0079, 0x0079,
                                0x0011, 0x0000, 0x0000, 0x0043,
                                0x0000, 0x0001 };
    conf[8] = fs;
    for (int i=0; i<10; i++) RegSet(i, conf[i]);
}
```

(b) AIC23_Base.cpp

EMIFの設定や，複数のクロック周波数を決める内蔵PLLの設定，およびボードの状態を設定するCPLDのレジスタの設定が行われています．

▶ **~Init6713DSK()**

デストラクタ~Init6713DSK()は定義からわかるように，何も行いません．なお，純粋仮想関数は派生クラスの側で定義するのが普通ですが，このデストラクタ~Init6713DSK()は何も行わないため，基底クラス側で定義しています．

● アナログ信号入出力のための基底クラス(**AIC23_Base**)

クラスAIC23_Baseは，クラスInit6713DSKを限定公開継承したものです．また，このクラスは基底クラス専用として使うため，抽象クラスとしました．

リスト2(a)にアナログ信号入出力のためのクラスAIC23_Baseに関するヘッダファイル(AIC23_Base.hpp)を，リスト2(b)に定義の本体(AIC23_Base.cpp)をそれぞれ示します．

(1)リスト2(a)，ヘッダについて

インクルードファイルcsl_mcbsp.hは，McBSPに関するCSLのAPIを使うためのヘッダです．

クラスの宣言の1行目で，「AIC23_Base」の後にある「: protected Init6713DSK」という記述は，クラスAIC23_BaseがクラスInit6713DSKを限定公開継承しているということを意味します．

▶ 公開メンバ

最初に，TLV320AIC23の標本化周波数を決めるためのシンボルが列挙型(enum)でAICfsというタグ名で宣言されています．

次のコンストラクタAIC23_Base()は，標本化周波数と2次キャッシュのサイズを決める二つの引き数を持っています．これらの引き数はデフォルトの引き数になっており，引き数を指定しない場合には，標本化周波数が48kHz，2次キャッシュのサイズが64Kバイトに設定されます．

RegSet()はTLV320AIC23内部のレジスタの値を設定するために使います．

次に宣言されているのはアナログ信号入出力のための関数です．TLV320AIC23とのデータの受け渡しでは，後で説明するように，McBSP1を使って32ビット単位で行います．この32ビットのデータは，左右2チャネルに対応する二つの16ビットデータで構成されています．そこで，データを受け取ってそれを二つのデータに分割するRead()，および二つのデータを一つのデータにまとめてデータを送るWrite()を使います．これらのメンバ関数は，引き数としてshort型およびfloat型の単純変数または配列に対応できるように，多重定義(オーバロード；overloading)されています．また，これらのメンバ関数は実行効率を高めるためインライン(inline)関数にしているので，その実装が同じリスト2(a)の中で定義されています．

▶ 限定公開メンバ

McBSP1のレジスタのアドレスが限定公開メンバとして宣言されています．このアドレスはこのクラスの，内部と派生クラスのみで使うので，限定公開としています．

▶ 非公開メンバ

最初にMcBSPに関する二つのハンドルが宣言されています．fsは標本化周波数を決める定数です．

次の，無名共用体（anonymous union）は，TLV320AIC23との間でデータのやり取りを行う際に32ビットデータと16ビットデータの変換を効率良く行うためのもので，この内部のメンバの対応関係は図6[注14]のようになっています．

Config()はTLV320AIC23内部のレジスタ0～9の値を一括して設定するために使います．

次の二つのメンバ関数は，TLV320AIC23との間で直接データをやり取りするために使います．

AIC23_Read()はTLV320AIC23でA-D変換されたデータを読み込む関数で，公開メンバ関数Read()の中で使います．この関数はA-D変換されたデータを読み取る際に，ポーリング方式と割り込み方式のどちらにも対応できるようにするため，ここでは純粋仮想関数として宣言しています．この関数の定義は，クラスAIC23_Baseを継承する派生クラスで行います．つまり，ポーリング方式はクラスAIC23_Pollingで，割り込み方式ではAIC23_Intrで定義を行います．

AIC23_Write()はTLV320AIC23にD-A変換のためのデータを書き込む関数で，公開メンバ関数Write()の中で使います．この関数は，ポーリング方式でも割り込み方式でも同じ定義のものを使うことができるので，仮想関数にはしていません．なお，実行効率を上げるため，ここではインライン関数として定義しています．

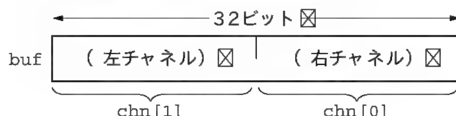
▶ インライン関数**Read()**，**Write()**の定義

Read()とWrite()は，それぞれ4通りの引き数に対応するようにオーバロードされています．これらの関数は，どれも関数名の先頭に「inline」を付け，インライン関数にします[注15]．TLV320AIC23のA-D/D-A変換器のビット幅が16ビットなので，これらのメンバ関数の引き数がshort型のときはデータの範囲が－32768～32767になります．一方，引き数がfloat型の場合は，データの範囲が－1以上1未満になるように正規化しています．つまり，A-D変換から読み込んだデータに対しては1/32768≅3.051758×10^{-5}を乗算し，D-A変換器に送るデータには32768を乗算しています．

McBSP1とのデータのやり取りは32ビット単位で行うため，32ビットデータと二つの16ビットデータ相互の変換が必要になります．関数Read()とWrite()ではその変換のために図6に示す共用体を使っているため，シフトやANDマスク操作が不要になり，実行効率を上げることができます．

引き数が配列の場合は，ch[0]が左チャネル，ch[1]が右チャネルに対応します[注16]．

〔図6〕クラスAIC23_Baseの非公開で宣言されている無名共用体のメンバの対応関係



（2）リスト2（b），定義本体について

▶**AIC23_Base()**

コンストラクタAIC23_Base()は，基底クラスの初期設定およびメンバ初期設定（member initialization）を行っています．コンストラクタ名の後の「:」（コロン）以下の部分がそのための記述です．InitC6713DSK(banks)は基底クラスの初期設定のための記述で，このクラスの基底クラスであるInitC6713DSKにbanksで指定された値を渡して，2次キャッシュのサイズを設定します．fs(fs_set)はメンバ初期設定のための記述で，標本化周波数を設定するために，このクラスの非公開メンバであるfsの値をfs_setで指定される値に設定します[注17]．

次に，TLV320AIC23とのインターフェースで使用するMcBSPのレジスタの設定を行います[注18]．この設定ではCSLのAPIを使っています．McBSP0はTLV320AIC23のレジスタの設定に使い，McBSP1はTLV320AIC23のA-D/D-A変換器とのデータのやり取りに使います．

McBSPのレジスタを設定するためのデータは，CSLが提供しているMCBSP_Config[注19]という構造体の型で宣言されているmcbsp0，mcbsp1に設定されます．McBSPの設定は，関数MCBSP_open()でオープンした後，関数MCBSP_config()で行っています．この設定で，McBSP0は送信データ長が16ビットに設定され，McBSP1は送信データ長および受信データ長が32ビットに設定されます．

その次に，関数MCBSP_start()でMcBSP0の動作を開始し，関数RegSet()でTLV320AIC23をリセットした後，TLV320AIC23のほかのレジスタを関数Config()で設定します．

最後に，関数MCBSP_start()でMcBSP1の動作を開始し

注14：なお，この関係はC6713 DSKのデフォルトであるリトルエンディアンモード（little-endian mode）の場合の対応関係になっている．ビッグエンディアンモード（big-endian mode）でDSKを使用するときは図6とは異なるので注意すること．

注15：正確には，コンパイラに対してインライン関数にできる条件が揃っていれば，インライン関数にしてもよいということを伝えるだけである．したがって，コンパイルされた結果が必ずインライン関数になるというわけではなく，コンパイラの最適化レベルによっては通常の関数呼び出しになる場合もある．

注16：共用体のメンバchn[]と引き数ch[]の間の対応関係は，chn[0]⇔ch[1]，chn[1]⇔ch[0]のようになっている．

注17：fsはconstメンバなので，これに値を代入することはできない．したがって，値を設定するためにはメンバ初期設定の構文を使う必要がある．

注18：この設定の部分は，C6713 DSKに付属するBSLのソースリストの一つであるdsk6713_aic23_opencodec.cを参考にした．なお，このファイルを見るためにはti¥c6000¥dsk6713¥libのdsk6713bsl.zipを展開する必要がある．

注19：参考文献4）の16章のp.7を参照のこと．

〔リスト3〕アナログ信号入出力をポーリング方式で行うための派生クラス(AIC23_Polling.hpp)

```
// ---------------------------------------------------------------------------
// AIC23_Baseの派生クラス(ポーリング方式用)
//  作成者:  三上直樹, 2003/08/11
// ---------------------------------------------------------------------------
#ifndef MK_AIC23_Polling

#include "AIC23_Base.hpp"

// AIC23_Pollingクラスの宣言, AIC23_Baseクラスを公開継承
class AIC23_Polling : public AIC23_Base
{
public:
    AIC23_Polling(const AICfs fs_set = fs48kHz, const L2mode banks = cache64K)
                  : AIC23_Base(fs_set, banks) {}
private:
// McBSP1の受信フラグを確認してTLV320AIC23からの32ビットのデータを読み込む
    unsigned int AIC23_Read() const
    {
        while ( (*(volatile unsigned int *)SPCR & 0x2) != 0x2);
        return *(unsigned int *)DRR;
    }
};

#define MK_AIC23_Polling
#endif
```

ます．なお，この段階で受信オーバランエラーが発生している可能性があります．このオーバランエラーは，McBSP1の受信レジスタを読み出すことにより解除できます．そこで，この関数の前後でMcBSP1の受信レジスタを読み出しています．

▶**RegSet()**

公開メンバ関数RegSet()は，第1引き数で指定されたTLV320AIC23の一つのレジスタに第2引き数で指定された値をMcBSP0を使って設定します．

▶**Config()**

非公開メンバ関数Config()は，McBSP0を使い，TLV320AIC23のレジスタ0～9の値を設定します．設定用のデータは配列conf[]にあります．この設定により，TLV320AIC23はA-D/D-A変換器のデータ長は16ビットに設定されます．また，ライン入力のゲインは＋6dBに設定されます．これは，DSKボードのライン入力ジャックとTLV320AIC23のライン入力端子の間に信号の大きさを1/2にする減衰器が入っているので，それを補正するための設定です．

● ポーリング方式によるアナログ信号入力のための派生クラス(**AIC23_Polling**)

クラスAIC23_Pollingは，クラスAIC23_Baseを公開継承したものです．クラスの宣言の1行目で,「AIC23_Polling」の後にある「 : public AIC23_Base」の記述は，クラスAIC23_PollingがクラスAIC23_Baseを公開継承しているということを意味します．

リスト3がヘッダファイル(AIC23_Polling.hpp)です．二つのメンバ関数は，どちらもインライン関数として定義しているので，定義本体のファイルはありません．

▶**AIC23_Polling()**

コンストラクタAIC23_Pollingは，引き数で与えられる標本化周波数と2次キャッシュのサイズに対する値を基底クラスに渡すだけで，それ以外の処理は行っていません．

▶**AIC23_Read()**

AIC23_Read()は，TLV320AIC23でA-D変換されたデータがMcBSP1に送られて来るまで待ち，データが送られて来たら，そのデータを取り込むためのメンバ関数です．この関数は，基底クラスの純粋仮想関数に対応するものです．

データが送られてきたことを確認する部分はwhile文になっていますが，SPCRに対するキャストの部分にvolatileというキーワードが使われていることに注意する必要があります．volatileが使われているのは，SPCRの値がプログラムでは変えられず，ハードウェアにより変えられるからです．このような場合に，volatileの指定がないと，コンパイラの最適化の影響で，while文になっているにもかかわらずSPCRの値を1度だけしか読み出さないというコードが生成される可能性があります．

このメンバ関数は，クラスAIC23_Pollingのユーザーからは直接呼ばれることはないため，非公開のメンバ関数にしています．

● 割り込みによるアナログ信号入力のための派生クラス(**AIC23_Intr**)

クラスAIC23_Intrは，クラスAIC23_Baseを公開継承したものです．リスト4(a)がヘッダファイル(AIC23_Intr.hpp)で，リスト4(b)が定義の本体(AIC23_Intr.cpp)です．

▶**IntrConfig**

構造体IntrConfigは，割り込み選択番号(割り込みイベントに与えられた番号)をCPU割り込みに割り当て，その割り込みを有効にするためのデータを作る際に使います．構造体のメンバはeventIDが割り込み選択番号に，intrNumberがCPU

〔リスト4〕アナログ信号入出力を割り込み方式で行うための派生クラス

```
// ----------------------------------------------------------
// AIC23_Baseの派生クラス(割り込み方式用)
//  作成者:  三上直樹, 2003/08/11
// ----------------------------------------------------------
#ifndef MK_AIC23_Intr

#include "AIC23_Base.hpp"

// AIC23_Intrクラスの宣言,AIC23_Baseクラスを公開継承
class AIC23_Intr : public AIC23_Base
{
public:
    struct IntrConfig { unsigned short eventID,
                                               intrNumber; };
    AIC23_Intr(const IntrConfig Table[],
               const AICfs fs_set = fs48kHz,
               const L2mode banks = cache64K);
private:
// McBSP1の受信フラグを確認せずにTLV320AIC23からの32ビットの
//                                              データを読み込む
    unsigned int AIC23_Read() const
                        { return *(unsigned int *)DRR; }
};

#define MK_AIC23_Intr
#endif
```

(a)AIC23_Polling.hpp

```
// ----------------------------------------------------------
// AIC23_Baseの派生クラス(割り込み方式用)
//  作成者:  三上直樹, 2003/08/11
// ----------------------------------------------------------
#include "AIC23_Intr.hpp"

AIC23_Intr::AIC23_Intr(const IntrConfig Table[],
                       const AICfs fs_set,
                       const L2mode banks) :
                       AIC23_Base(fs_set, banks)
{
    int n = 0;
    while(Table[n].intrNumber != 0)
    {
        IRQ_map(Table[n].eventID, Table[n].intrNumber);
        IRQ_clear(Table[n].eventID);
        IRQ_enable(Table[n].eventID);
        n++;
    }
    IRQ_nmiEnable(); ←ノンマスカブル割り込み(NMI)を許可
}
```

(b)AIC23_Polling.cpp

割り込み番号にそれぞれ対応します.この構造体はコンストラクタ`AIC23_Intr`で使われます.

▶**`AIC23_Intr()`**

コンストラクタ`AIC23_Intr()`は,引き数で与えられる標本化周波数と2次キャッシュのサイズに対する値を基底クラスに渡します.

次に,このコンストラクタの第1引き数で与えられる構造体`IntrConfig`を要素とする配列の値に基づいて,割り込みに関する設定を行います.関数`IRQ_map()`は割り込み選択番号をCPU割り込みに割り当てます.次に,関数`IRQ_clear()`で対応する割り込みフラグをクリアした後,関数`IRQ_enable()`で対応するマスカブル割り込みの割り込みマスクを設定して,その割り込みを許可します.これを構造体`IntrConfig`の配列要素のメンバ`intrNumber`が0になるまで続けます.

最後に,関数`IRQ_nmiEnable()`で,NMI(Non Maskable Interrupt)を許可します[注20].

割り込みを使うためには,そのほかにグローバル割り込みを許可する必要があります.そのため,CSLのAPIでは関数`IRQ_globalEnable()`がサポートされています.しかし,このコンストラクタが実行された後,割り込みを許可する前にほかの処理が必要になる場合もあるので,コンストラクタの中ではグローバル割り込みを許可する処理は行いません.

▶**`AIC23_Read()`**

割り込みを使う場合は,TLV320AIC23でA-D変換されたデータがMcBSP1に送られて来るまで待つ必要はありません.そこで,このメンバ関数は直ちにデータを取り込みます.この関数は,基底クラスの純粋仮想関数に対応するものです.

このメンバ関数は,クラス`AIC23_Intr`のユーザーからは直接呼ばれることはないため,非公開のメンバ関数にしています.

おわりに

今回作成したクラスで,ユーザーがアクセス可能なメンバの一覧を表2に示します.

今回作成したクラスのソースをコンパイルしてライブラリを作る方法,およびこのライブラリを使ったプログラムの例は次回に説明します.

プログラムのソースリストのファイルはInterGiga No.33に収録します.なお,Code Composer Studio(CCS)は日本語に対応していないため,収録されるソースリストのコメントは英語で書かれています.

参考文献

1)三上 直樹,『C言語によるディジタル信号処理入門』,CQ出版社,2002年.

2)『TMS320C6713 DSK Technical Reference』,Spectrum Digital,2003年5月.

3)『TLV320AIC23 Data Manual』,文献番号SLWS106C,Texas Instruments,2001年7月.

4)『TMS320C6000 Chip Support Library API User's Guide』,文献番号SPRU401F,Texas Instruments,2003年2月.

みかみ・なおき 職業能力開発総合大学校 情報工学科

注20:TMS320C6000シリーズのDSPでは,NMIを許可しないかぎり,他のマスカブル割り込みは無効になっている.したがって,マスカブル割り込みを使う場合は,NMIを使わない場合であってもNMIを許可する必要がある.

〔表2〕クラスのユーザーがアクセス可能なメンバの一覧(メンバ関数の戻り値はすべてなし)

構　文	AIC23_Polling(const AICfs fs_set = fs48kHz, const L2mode banks = cache64K);	
機　能	ポーリング方式でアナログ信号入力を行う際に使うクラスのコンストラクタ	
引き数	fs_set	標本化周波数を決める定数
	banks	2次キャッシュのサイズを決める定数
定義の場所	AIC23_Polling	
タイプ	コンストラクタ	
構　文	AIC23_Intr(const IntrConfig Table[], const AICfs fs_set = fs48kHz, const L2mode banks = cache64K);	
機　能	割り込み方式でアナログ信号入力を行う際に使うクラスのコンストラクタ	
引き数	Table[]	有効な割り込みを設定する配列
	fs_set	標本化周波数を決める定数
	banks	2次キャッシュのサイズを決める定数
定義の場所	AIC23_Intr	
タイプ	コンストラクタ	
構　文	void Read(short &chL, short &chR); void Read(float &chL, float &chR); void Read(short ch[]); void Read(float ch[]);	
機　能	TLV320AIC23からの信号の読み込み	
引き数	&chL	左チャネルの信号
	&chR	右チャネルの信号
	ch[]	ch[0]：左チャネルの信号，ch[1]：右チャネルの信号
定義の場所	AIC23_Base	
タイプ	メンバ関数	
構　文	void Write(const short chL, const short chR); void Write(const float chL, const float chR); void Write(const short ch[]); void Write(const float ch[]);	
機　能	TLV320AIC23への信号の書き込み	
引き数	chL	左チャネルの信号
	chR	右チャネルの信号
	ch[]	ch[0]：左チャネルの信号，ch[1]：右チャネルの信号
定義の場所	AIC23_Base	
タイプ	メンバ関数	
構　文	void RegSet(const unsigned short reg, const unsigned short val);	
機　能	TLV320AIC23のレジスタの設定	
引き数	reg	レジスタ番号(0～9，15)
	val	設定する値
定義の場所	AIC23_Base	
タイプ	メンバ関数	
宣　言	struct IntrConfig { unsigned short eventID, intrNumber; };	
説　明	有効な割り込みの設定のため，コンストラクタのAIC23_Intr()の引き数として使用	
メンバ	eventID	割り込み選択番号
	intrNumber	CPU割り込み番号
定義の場所	AIC23_Intr	
タイプ	構造体	
宣　言	enum AICfs { fs96kHz = 0x001D, fs48kHz = 0x0001, fs44kHz = 0x0023,fs32kHz = 0x0019, fs24kHz = 0x0041, fs16kHz = 0x0059, fs08kHz = 0x000D };	
説　明	標本化周波数設定のための定数	
定義の場所	AIC23_Base	
タイプ	列挙型	
宣　言	enum L2mode { cache0K, cache16K, cache32K, cache48K, cache64K = 7 };	
説　明	2次キャッシュのサイズ設定のための定数	
定義の場所	InitC6713DSK	
タイプ	列挙型	

XScaleプロセッサ徹底活用研究

第5回　CPUローカルバスの制御方法とPCIバスブリッジの実装

山武 一朗

前回(本誌2003年12月号掲載)はXScale評価キットにCompactFlashソケットを実装した．今回はより拡張性の高い，PCIバスへのブリッジを実現する．まず，PXA25xのローカルバスの制御方法について解説したあと，外付けにPCIバス形状のFPGA評価ボードを接続し，このFPGA内にPCIバスブリッジを実装することで，PXA25xをPCIバスにアクセスできるようにする．（編集部）

はじめに

PXA25xのローカルバスは，ごく一般的なマイコンのバスとなっています．バス幅は16ビット幅と32ビットのどちらかしか選択できませんが，各種アクセス速度に対応できるよう，アクセス速度の設定範囲が広く，またタイミングによってアクセス速度が異なる，ウェイトを要求するデバイスも接続することができます．

今回はこのローカルバスを活用するために，スタティックメモリエリアの機能について解説したあと，FPGA評価キットであるStratix評価キットと接続し，PCIバスへのブリッジを設計してみます．

なお，XScale評価キットはもちろん，Stratix評価キットの基本的な操作方法などについてはここでは説明しません．とくにStratixなどのFPGA開発ツールなどについては，各キットの取扱い説明書や参考記事をよく読んで，操作方法について慣れておく必要があると思います．

〔図1〕PXA25x/PXA26xのメモリマップ

アドレス	領域
0xFFFFFFFF	予約
0xB0000000	SDRAMバンク SDCS3(64Mバイト)
0xAC000000	SDRAMバンク SDCS2(64Mバイト)
0xA8000000	SDRAMバンク SDCS1(64Mバイト)
0xA4000000	SDRAMバンク SDCS0(64Mバイト)
0xA0000000	予約
0x4C000000	内蔵レジスタ
0x40000000	PCMCIA/CFスロット0(256Mバイト)
0x30000000	PCMCIA/CFスロット1(256Mバイト)
0x20000000	予約
0x18000000	スタティックメモリエリア CS5(64Mバイト)
0x14000000	スタティックメモリエリア CS4(64Mバイト)
0x10000000	スタティックメモリエリア CS3(64Mバイト)
0x0C000000	スタティックメモリエリア CS2(64Mバイト)
0x08000000	スタティックメモリエリア CS1(64Mバイト)
0x04000000	スタティックメモリエリア CS0(64Mバイト) StrataFlash memory(PXA26x)
0x00000000	

1 PXA25xのローカルバス機能概要

● メモリマップ

図1にPXA25xのメモリマップを示します．連載第1回(2003年6月号)でも解説したように，PXA25xにはSDRAMやフラッシュメモリなど，さまざまなメモリを直接接続することができます．その中で，今回使用するのは，スタティックメモリエリアと呼ばれる空間です．

スタティックメモリエリアには，ROMやSRAM，一般的なI/Oデバイスなど，アドレスバスをマルチプレクスしないデバイスを接続できます．このスタティックメモリエリアのバスアクセスタイミングを設定するレジスタとして，MCS0～2が用意されています．

● MCxレジスタ

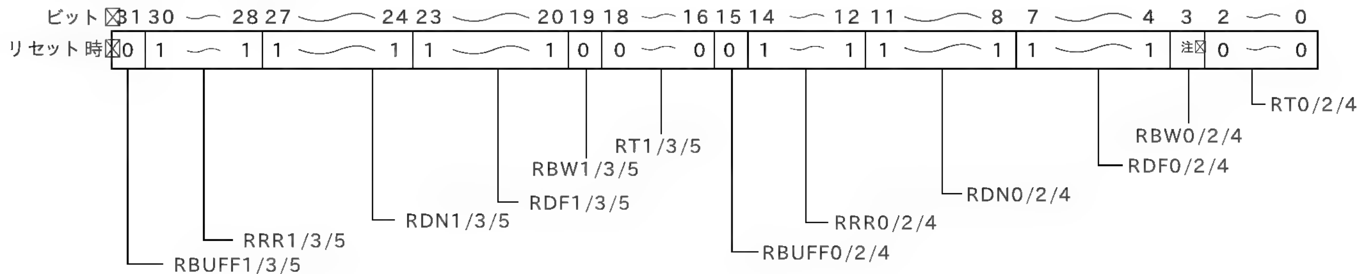
図2にMCS0～2レジスタのフォーマットを示します．スタティックメモリエリアはエリア0～5まで六つあるわけですが，MCS0～2は32ビット長のレジスタを上位16ビットと下位16ビットにわけて6エリア分の設定を行います．

RTxビット(ビット2～0)は，そのエリアに接続するメモリの種類を選択します．表1に接続できるメモリの種類と設定値を示します．通常のROMやRAM，4ワードまたは8ワードバーストアクセス可能なROM，可変レイテンシI/Oのデバイスの計5種類となっています．

RBWxビット(ビット3)のはバス幅の設定で，0で32ビット幅，1で16ビット幅の設定になります．

RDFxビット(ビット7～4)はチップセレクトをアサートして最初のワードデータを読み出すまでのディレイ時間を設定します．RDNxビット(ビット11～8)はバーストアクセス時の2ワード目以降のディレイ時間を設定します．RRRxビット(ビット14～12)はリカバリタイムを設定します．RBUFFxビット(ビット15)は，低速のデバイスを接続する場合は0，高速デバ

〔図2〕MCSxレジスタのフォーマット



注：エリア0ではBOOT_SELピンの設定による．それ以外のエリアでは0

イスを接続する場合は1に設定します．

● バスアクセスタイミング

図3(p.148)に基本的なローカルバスのバスアクセスタイミングを示します．

図3(a)は4ワードバーストアクセス対応のROMにアクセスした場合の例です．MCxレジスタの設定は，RT＝2(4ワードバーストROM)，RDF＝4，RDN＝1，RRR＝0です．図3(b)は可変レイテンシのI/Oデバイスにアクセスした場合の例です．MCxレジスタの設定は，RT＝4(可変レイテンシI/O)，RDF＝2，RDN＝2，RRR＝1です．RDY信号はメモリクロックで2クロック期間"H"レベルを認識した時点で，ウェイト完了と見なされます．

● 各デバイス接続時における最小タイミング

MCSxレジスタでアクセスタイミングを設定しますが，タイミングとして0クロックを設定しても，場合によっては自動的に最小アクセスタイミングとしてウェイトが挿入されることがあります．

表2に各種デバイス接続時のアクセスタイミングを示します．アドレスのアサートやリード/ライトクロックのアサートには1クロックが加算されます．可変レイテンシI/Oでは，最初のアクセスにRDFとRDNのクロックを足したうえに，さらに2クロックが加算される計算になります．可変レイテンシI/Oの設定では，たとえ外部からのウェイト要求がなくても，かなりのウェイトが挿入されてしまうことになります．

● 可変レイテンシI/O時の書き込み信号

通常の書き込み信号としてはnWE信号がありますが，PXA25xでは，可変レイテンシI/Oを接続している場合の書き込み信号としては，nPWEを使うように規定されています．nPWEは連載第4回(2003年12月号)でも解説した，PCカード/CFカード接続時に使う信号ですが，PCカードやCFカード以外であっても，ウェイトを要求するデバイスを接続する場合は書き込み信号としてこの信号を使います．

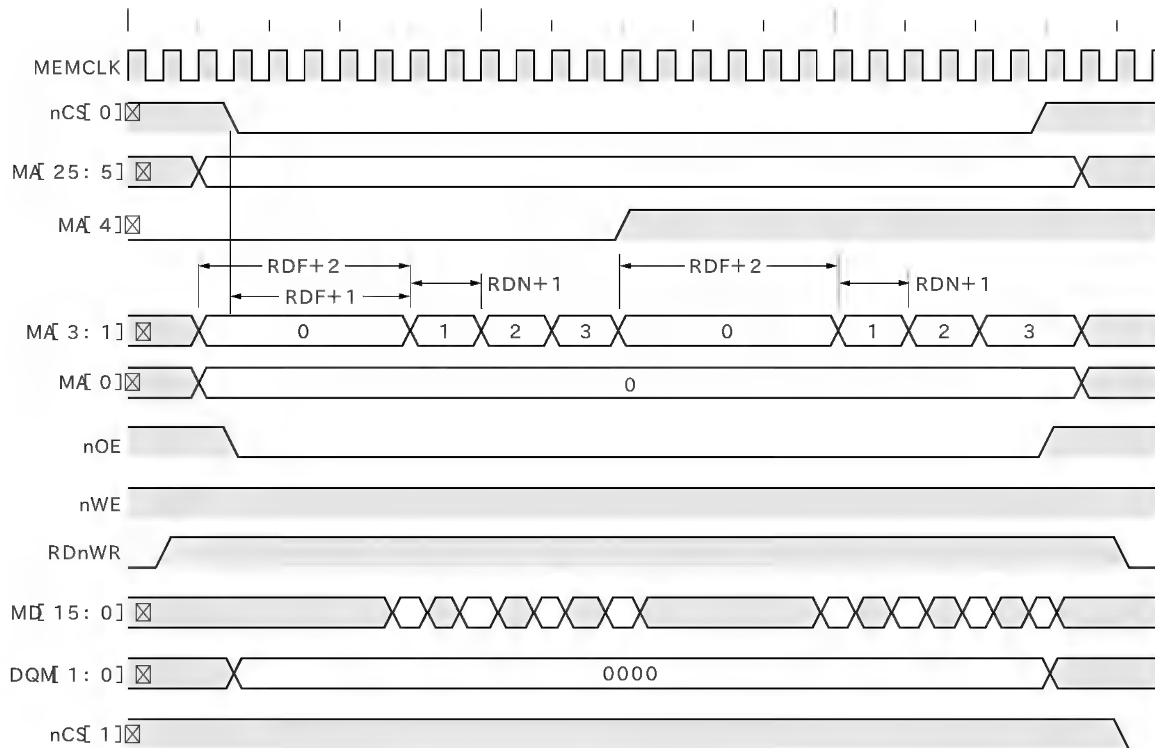
〔表1〕MCxレジスタのRTxビットの設定値

RTx(ビット2〜0)	デバイスタイプ
000	バースト非対応ROM，またはフラッシュメモリ
001	SRAM
010	4ワードバースト対応ROM，またはフラッシュメモリ(書き込みは非バースト)
011	8ワードバースト対応ROM，またはフラッシュメモリ(書き込みは非バースト)
100	可変レイテンシI/O
101	予約
110	予約
111	予約

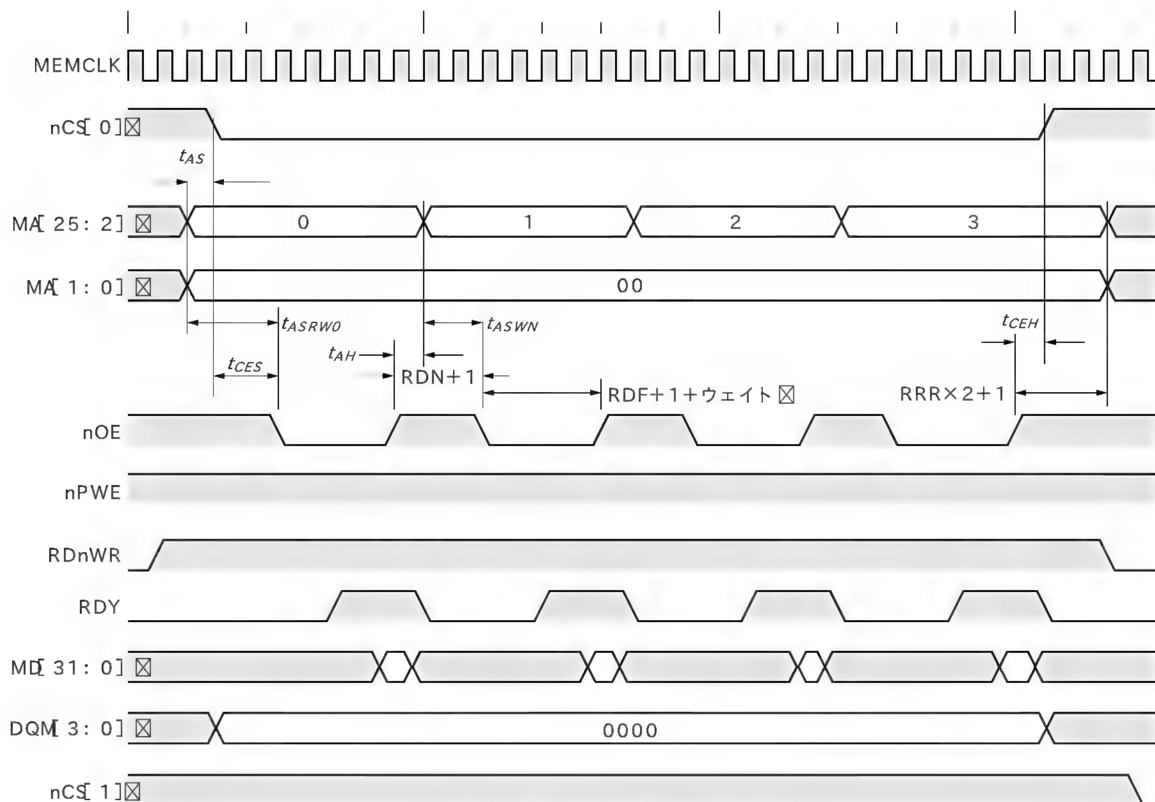
〔表2〕接続デバイスとリード/ライトタイミング

デバイスタイプ	アクセスタイミング					
	バーストリードアドレスアサート	nOEアサート	バーストnOEディアサート	バーストライトアドレスアサート	nWEアサート	バーストnWEディアサート
非バーストROMまたはフラッシュメモリ	RDF+1	RDF+1	0	—	RDF+1	—
SRAM	RDN+1	RDN+1	0	RDN+2	RDN+1	1
4ワードバーストROMまたはフラッシュメモリ(非バーストライト)	RDF+1 (0，4) RDN+1 (1：3，5：7)	RDF+1 (0，4) RDN+1 (1：3，5：7)	0	—	RDF+1	—
8ワードバーストROMまたはフラッシュメモリ(非バーストライト)	RDF+1 (0) RDN+1 (1：7)	RDF+1 (0) RDN+1 (1：7)	0	—	RDF+1	—
可変レイテンシI/O	RDF+RDN+2+ウェイト	RDF+1+ウェイト	RDN+1	RDF+RDN+2+ウェイト	RDF+1+ウェイト	RDN+1

〔図3〕バスアクセスタイミング



(a) 4ワードバーストアクセスのROM接続時



(b) 可変レイテンシI/Oのデバイス接続時

なお，nPWEに関しては，PCカード/CFカードインターフェース機能をイネーブルにする信号（MECRレジスタのCITビット）をセットしなくても，書き込み信号が出力されるようです．

● バーストアクセスの発生するタイミング

図3のアクセスには，どちらもバーストアクセスが行われているようすが示されています．バーストアクセス時はチップセレクトがアサートされっぱなしになることがわかります．ローカルバスに接続するデバイスがバーストアクセスをサポートする場合は，CPUが今何ワード目を読み込んでいる（書き込んでいる）かを把握しながら動作する必要があります．つまり，メモリクロックに同期した同期設計が必要になります．

● 忘れがちなGPIOの設定

PXA25xのローカルバスを使ううえでよく忘れるのが，GPIOの設定です．アドレスバスやデータバス，チップセレクト，OE信号やWE信号のリード/ライト信号は，それぞれ専用のピンが割り当てられているので問題ありません．注意が必要なのは，外部からウェイトを要求するためのRDY信号と，可変レイテンシI/O使用時のライト信号であるnPWE信号です．これらの信号は，汎用I/OであるGPIOと兼用ピンに割り当てられていて，しかもリセット直後のデフォルトの設定では，GPIO機能が選択されています．

具体的には，RDY信号はGPIOのGP18と，nPWE信号はGPIOのGP49と兼用ピンになっているので，ファンクション設定レジスタGAFR0_Uのビット5～4を'01'にしてRDY機能を，GAFR1_Uのビット3～2を'10'にしてnPWE機能を有効にします．さらにGPIOの入出力方向設定レジスタGPDR0のビット18をクリア（入力方向），GPDR1のビット17をセット（出力方向）して，各ピンの入出力方向も正しく設定します．

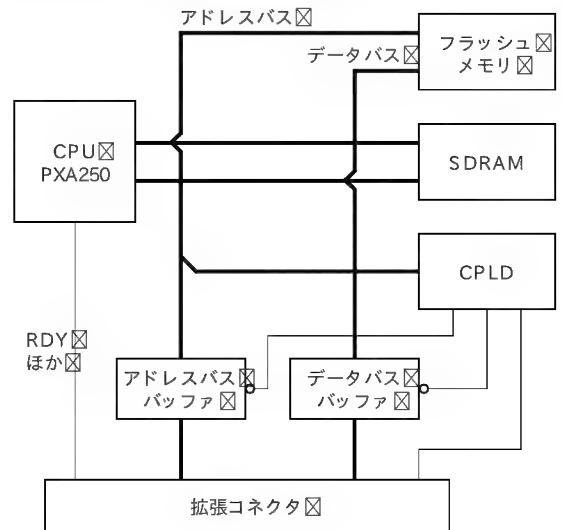
2 評価ボードでのローカルバス使用上の注意

● 評価ボードの拡張コネクタについて

図4に評価ボードのローカルバスと拡張コネクタ周辺のブロック図を示します．図を見るとわかるように，評価ボード上のフラッシュメモリやSDRAMはCPUのアドレスバスやデータバスが直結されていますが，拡張コネクタに配線されているアドレスバスやデータバスは，バスバッファを経由した信号である点に注意が必要です．これは，拡張コネクタに接続されるデバイスの容量負荷などの変化で，クロック周波数100MHzで駆動するオンボードのSDRAMの動作が不安定になるのを防ぐことを考慮したものだと思います．

問題は，CPUから出力されている3本のクロック信号SDCLK0～2も，バスバッファを経由して拡張コネクタに接続されている点です．つまり，クロックがバスバッファを経由しているので，拡張コネクタに届いた時点ではCPUの動作と同期が取れたクロックであるとはいいがたいわけです．さらに，可変レイテンシI/Oを接続する場合に必須となるRDY信号は，

〔図4〕評価ボードの拡張コネクタ周辺のブロック図



バスバッファを経由せず，拡張コネクタから直接CPUに接続されています．

● 非同期バスと想定して設計

以上のことから，拡張コネクタに配線されているクロックとRDY信号を使って，CPUのローカルバスに接続するデバイスを同期設計で実現することは難しいのではないかと思われます．そこで，拡張コネクタの部分は非同期バスであるとしてローカルバス部分を設計しました．

CPU側からみると，外部から入力されるRDY信号も非同期信号になるわけなので，外部デバイスが返したRDYをCPUがどのタイミングで取り込んでくれるかはわかりません．また，アドレスバスやローカルバスがバスバッファを経由していることで，バスのタイミングに若干の注意が必要です．

まず書き込み方向について考えます．ローカルバスの信号は基本的にはRDY信号以外はすべてバスバッファを経由しています．すべてのバスバッファは同じ種類のデバイスなので，基本的にディレイ時間は同じと考えると，書き込み方向については，すべての信号が一様に遅延して拡張コネクタに到達すると考えられます．そこで書き込み信号がアサートされたことを確認したら，データバス上の書き込みデータを取り込み，RDYを返します．ここで返したRDYをどのタイミングでCPUが認識するのかはわかりませんが，少なくとも書き込み信号がディアサートされれば1ワード分の書き込み動作は完了したと判断すればよいでしょう．

次は読み出し方向について考えます．アドレスバスや制御信号は一様に遅れて拡張コネクタまで到達しますが，読み出しデータ，つまり外部デバイスから見た出力データはバスバッファを経由してCPUに到達することになります．つまりデータバスの遅延を考慮してRDYを返さなければならないわけです．そこで，読み出しデータをデータバス上に出力したらRDY

を返しますが，データバスはリード信号であるnOE信号がディセーブルになるまでドライブするようにしました．nOE信号のアサート時間はバスバッファの遅延を考慮して若干長めに設定しておくのが安全でしょう．またCPUがnOE信号をディセーブルにしても，バスバッファを経由してそれが外部デバイスに伝わるので，外部デバイスが実際にデータバスを開放するタイミングが遅れますが，それも考慮してMCSxレジスタに設定するリカバリタイムなどを設定すれば，バスが衝突することはないでしょう．

また，実際には，CPUボード上のバスバッファは，CPUのリード/ライト方向でバスのドライブ方向を制御するので，拡張コネクタより外側のデバイスがデータバスをだらだらとドライブし続けても，CPUボード上のデータバスはバスバッファにより切り離され，動作に問題はありません．

3 FPGA内部レジスタの読み書きテスト

● Stratix評価キットとの接続

それでは，PXA 25xのローカルバスのテストのために，本RISC評価キットシリーズ同様，CQ出版から発売されているFPGA評価キットの中のStratix評価キットを，本評価ボードのローカルバスに接続してみましょう．

Column

CompactFlashソケット拡張キット

CQ RISC評価キット/XScale（CQ出版）に，CompactFlashソケット（以下CFソケット）やFPGAを接続するための拡張キットが用意されました．

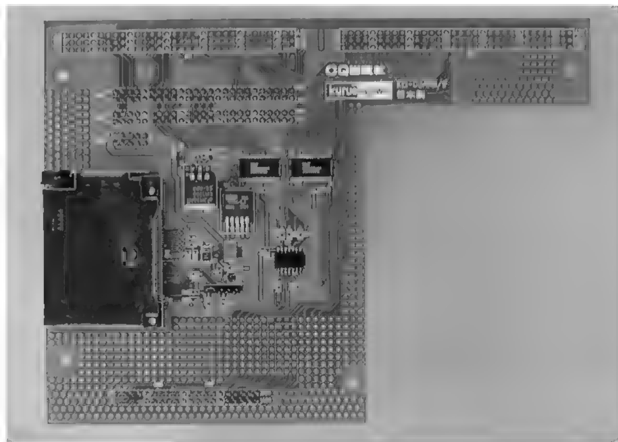
XScale評価ボードにCFソケットを実装した場合は，写真AのCFソケット拡張キットの上にXScale評価ボードをスタック接続し，写真Bのようにして使います．

XScale評価ボードにFPGAを接続する場合，FPGA評価キットとして次の2種類のキットが対応しています．一つはStratix EP1S10（アルテラ）を搭載した「Stratix評価キット」（http://www.cqpub.co.jp/eda/Stratix/）と，Spartan-IIE XC2S300E（ザイリンクス）を搭載した「Spartan-IIE300評価キット」（http://www.cqpub.co.jp/eda/Spartan2e300/）です．どちらでも使い慣れたほうのFPGAを使用することができます．これらFPGA評価キットのどちらかの評価ボードを1枚用意し，その上にCFソケット拡張キットを乗せ，さらにその上にXScale評価ボードをスタック接続します（写真C）．

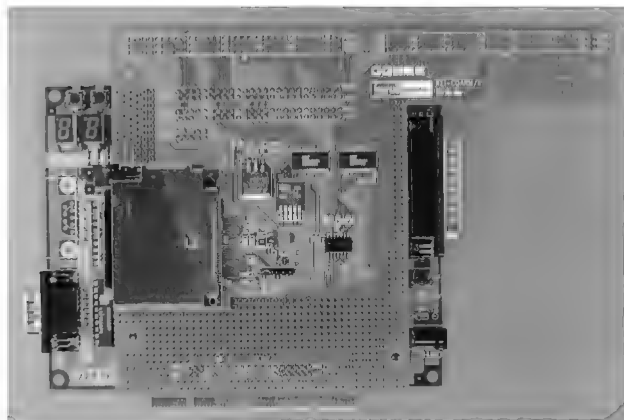
CF拡張キットは，下記URLに詳細情報を掲載しています．購入方法についても，下記URLを参照してください．

■問い合わせ先
- 来栖川電工（有）
  http://www.kurusugawa-ele.co.jp/cq/cfkit.html
- 組み込みshop　http://shop.kumikomi.net/webmall/

〔写真A〕CFソケット拡張キット



〔写真B〕Stratix評価ボードと接続したようす



〔写真C〕Stratix評価ボードとXScale評価ボードを接続したようす

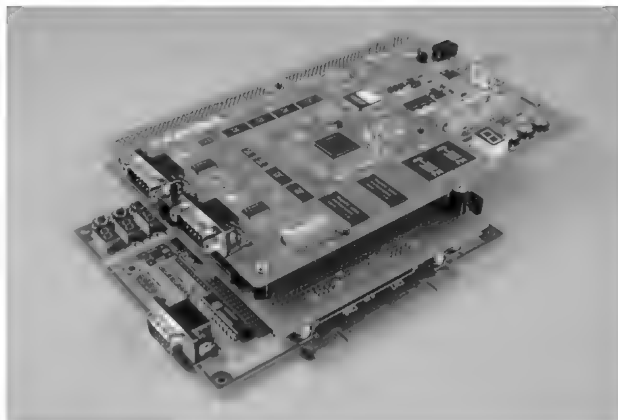


図5にCPUボードとStratixボードの接続図を示します．接続する相手方はFPGAなので，ピン配置自体にはあまり意味はないでしょう．PXA25xのローカルバスとして必要な信号をFPGAのI/Oピンに接続するだけです．スタティックメモリエリアのチップセレクトは，エリア0とエリア2以外を配線しました．エリア0にはフラッシュメモリが，エリア2にはLEDやディップスイッチの入出力レジスタなどがマッピングされ，CPUボード上ですでに使われているためです．

なお，可変レイテンシI/Oデバイスとして接続するので，書き込み信号は$\overline{\text{PWE}}$を使います．また，PCIブリッジからの割り込み信号をCPUに入力するために，GPIO10をFPGAと配線しておきました．

写真1に手配線で試作した両キットをスタック接続するボードの外観を示します．写真1(a)のボードを，写真1(b)のようにStratixボードの上にのせ，さらにその上に写真1(c)のようにCPUボードを接続します．

● FPGA内部に読み書きレジスタを実装

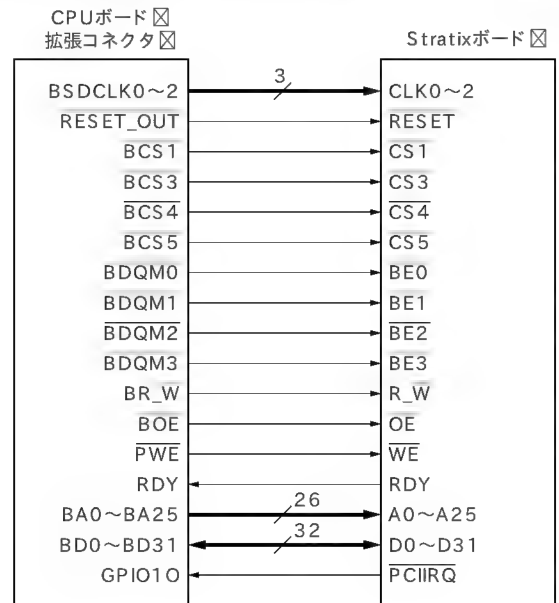
まず，CPUとFPGAの間で，正しくデータの読み書きができるかどうかをテストする必要があります．ここでは，FPGAの内部に読み書き可能なテストレジスタを実装し，CPUから正しくアクセスできるかどうかを確認します．

リスト1に，FPGA内部に読み書きテストを行えるテストレジスタを実装したVHDLソースのサンプルを示します．FPGA内部に4本のテストレジスタを実装し，読み書きできるようにしてみました．また，アクセスするたびに内部のウェイトカウンタがインクリメントし，RDYを返すタイミングを変えるようなハードウェアになっています．

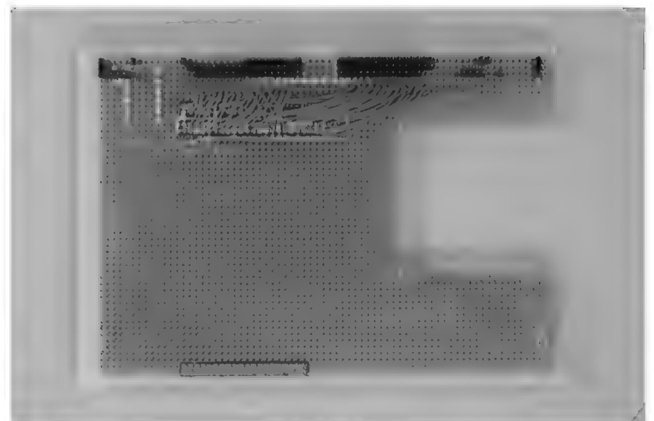
テストハードウェアとはいえ，後述するPCIブリッジでもローカルバス制御部を流用できるよう，階層設計にしてみました．

ローカルバス制御部では，チップセレクトやリード/ライト信号がアサートされるのを待ちます．これらの信号がアサートされたら，バスアクセススタート信号をセットします．また，ライト時は書き込みデータをローカルデータバスから取り込んで

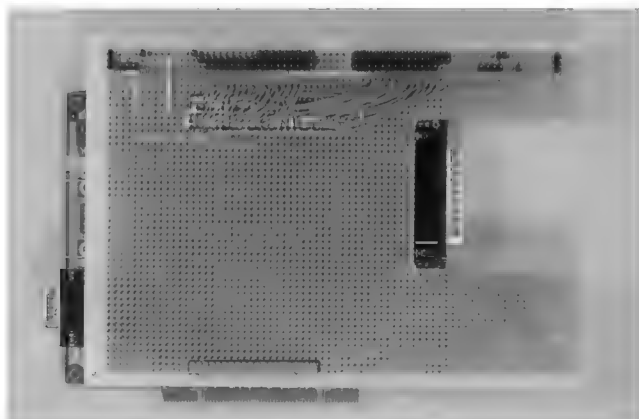
〔図5〕CPUボードとStratixボードの接続図



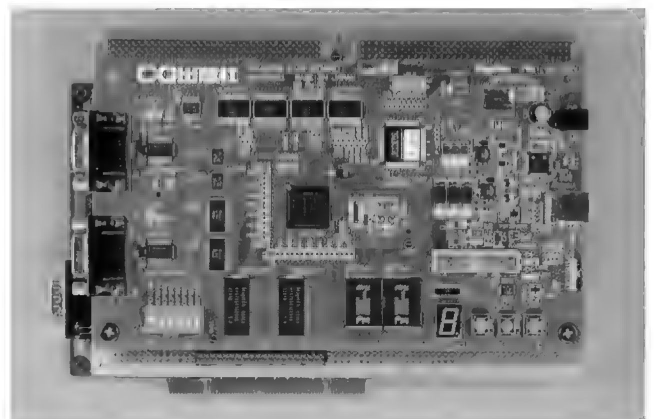
〔写真1〕試作した接続ボード



(a) スタック接続ボード単体



(b) Stratixボードの上に接続



(c) さらに上にCPUボードを接続

〔リスト 1〕FPGA 内部にテストレジスタを実装した VHDL ソース(一部)

```
LOCAL_Seq : process( CLK, nRST )

    -- ステート変数
    variable CSTATE : std_logic_vector(1 downto 0);
    variable NSTATE : std_logic_vector(1 downto 0);
    -- ステート定義
    constant BUS_IDLE       : std_logic_vector(1 downto 0) :="00";
    constant ACC_WAIT       : std_logic_vector(1 downto 0) :="01";
    constant TURN_AROUND    : std_logic_vector(1 downto 0) :="11";

begin

    if (nRST = '0') then    -- リセット

         CSTATE := BUS_IDLE;
         NSTATE := BUS_IDLE;
         DATA_OD   <= (others => '0');
         RDY       <= '0';
         WriteBuff <= (others => '0');
         AccStart  <= '0';

    elsif (CLK'event and CLK = '1') then

         CSTATE := NSTATE;  -- ステートマシン制御
         case CSTATE is

    -- ********** BUS_IDLE時の動作 ********** --
         when BUS_IDLE =>    -- バスアイドル時

              if (nHit = '0') then         -- デバイス選択時
                   if (nRD = '0') then      -- リードアクセス時
                        RDY <= '0';         -- ローカルバスウェイト要求
                        AccStart  <= '1';   -- 内部アクセススタート
                        NSTATE := ACC_WAIT;
                   elsif (nWR = '0') then  -- ライトアクセス時
                        WriteBuff <= DATA;  -- 書き込みデータ取り込み
                        RDY <= '0';         -- ローカルバスウェイト要求
                        AccStart  <= '1';   -- 内部アクセススタート
                        NSTATE := ACC_WAIT;
                   else
                        NSTATE := BUS_IDLE;
                   end if;
              else
                   NSTATE := BUS_IDLE;
              end if;

    -- ********** ACC_WAIT時の動作 ********** --
         when ACC_WAIT =>    -- アクセス完了待ち

              AccStart <= '0';         -- 内部アクセススタート信号クリア
              if (AccReady = '1') then-- 内部アクセス完了
                   if (nRD = '0') then-- リードアクセス時
                        -- リードバッファをローカルデータバスバッファへ転送
                        DATA_OD <= ReadBuff;
                   end if;
                   RDY <= '1';         -- ローカルバスレディ
                   NSTATE := TURN_AROUND;
              else
                   NSTATE := ACC_WAIT;
              end if;

    -- ********** TURN_AROUND時の動作 ********** --
         when TURN_AROUND =>      -- ターンアラウンドステート
              -- リード/ライトクロックがディアサートされたら
              if (nRD = '1' and nWR = '1') then
                   RDY    <= '0';
                   NSTATE := BUS_IDLE;
              else
                   NSTATE := TURN_AROUND;
              end if;

    -- ****************************************** --
         when others =>
              AccStart <= '0';    -- 内部アクセススタート信号クリア
              RDY      <= '1';    -- ローカルバスレディ
              NSTATE := BUS_IDLE;

         end case;

    end if;

end process LOCAL_Seq;
```

(a) PXA25x ローカルバスシーケンサ

```
TEST : process(Clock, Reset)
                                        ～中略～
    -- アクセスカウンタ
    variable ACount : std_logic_vector(4 downto 0);
    -- ウェイトカウンタ
    variable WCount : std_logic_vector(4 downto 0);

begin

    if (Reset = '1' ) then
                                        ～中略～
        ReadBuff  <= (others => '0');
        AccReady  <= '0';
        ACount     := (others => '0');
        WCount     := (others => '0');

    elsif (Clock'event and Clock = '1') then

        CSTATE := NSTATE;  -- ステートマシン制御
        case CSTATE is

        -- ********** IDLE時の動作 ********** --
             when IDLE =>   -- アイドル時
                  AccReady <= '0';
                  WCount    := (others => '0');
                  if (AccStart = '1') then
                       if (ReadWrite = '1') then -- リードアクセス時
                            NSTATE := READ;
                       else                       -- ライトアクセス時
                            NSTATE := WRITE;
                       end if;
                  else
                       NSTATE := IDLE;
                  end if;

        -- ********** READ時の動作 ********** --
             when READ =>   -- リードアクセス時
                  case Address(7 downto 2) is
                  when "000000" =>
                       ReadBuff <= TEST_REG0;
                  when "000001" =>
                       ReadBuff <= TEST_REG1;
                  when "000010" =>
                       ReadBuff <= TEST_REG2;
                  when "000011" =>
                       ReadBuff <= TEST_REG3;
                  when others =>
                       ReadBuff <= (others => '0');
                  end case;
                  NSTATE := COMP;

        -- ********** WRITE時の動作 ********** --
             when WRITE =>  -- ライトアクセス時
                  case Address(7 downto 2) is
                  when "000000" =>
                       if (ByteEnable(3) = '1') then
                            TEST_REG0(31 downto 24) <=
                                      WriteBuff(31 downto 24);
                       end if;
                       if (ByteEnable(2) = '1') then
                            TEST_REG0(23 downto 16) <=
                                      WriteBuff(23 downto 16);
                       end if;
                       if (ByteEnable(1) = '1') then
                            TEST_REG0(15 downto  8) <=
                                      WriteBuff(15 downto  8);
                       end if;
                       if (ByteEnable(0) = '1') then
                            TEST_REG0( 7 downto  0) <=
                                      WriteBuff( 7 downto  0);
                       end if0;
                       ～中略～
                  when others =>
                       null;
                  end case;
                  NSTATE := COMP;

        -- ********** COMP時の動作 ********** --
             when others => -- アクセスウェイト
                  if (ACount = WCount) then
                       ACount := ACount + '1'; -- アクセスカウンタ+1
                       AccReady <= '1';         -- アクセス完了
                       NSTATE := IDLE;
                  else
                       WCount := WCount + '1'; -- ウェイトカウンタ+1
                       NSTATE := COMP;
                  end if;

        end case;

    end if;

end process TEST;
```

(b) テストレジスタアクセスシーケンサ

〔リスト 2〕バス制御レジスタの初期化例

```
void Bus_Init(void)
{
    *MCS0    = (*MCS0    & 0x0000FFFF) | (0x0774 << 16); /* エリア1設定 */
    *GAFR0_u = (*GAFR0_u & 0xFFFFFFCF) | 0x10;  /* GP18 RDY信号 */
    *GPDR0   = (*GPDR0   & 0xFFFBFFFF);         /* RDY信号(GP18) 入力方向*/
    *GAFR1_u = (*GAFR1_u & 0xFFFFFFF3) | 8;     /* GP49 PWE信号 */
    *GPDR1   = *GPDR1 | 0x20000;                /* PWE信号(GP49) 出力方向*/
}
```

〔表 3〕PCIブリッジのメモリマップ

スタティックメモリエリア	アドレス	用途
エリア1	0x04000000 ～ 0x07FFFFFF	SDRAM空間
エリア3	0x0C000000 ～ 0x0DFFFFFF	PCI制御レジスタ空間
	0x0E000000 ～ 0x0FFFFFFF	PCI I/O空間
エリア4	0x10000000 ～ 0x10FFFFFF	PCIメモリウィンドウ0
	0x11000000 ～ 0x11FFFFFF	PCIメモリウィンドウ1
	0x12000000 ～ 0x12FFFFFF	PCIメモリウィンドウ2
	0x13000000 ～ 0x13FFFFFF	PCIメモリウィンドウ3
エリア5	0x14000000 ～ 0x14FFFFFF	PCIメモリウィンドウ4
	0x15000000 ～ 0x15FFFFFF	PCIメモリウィンドウ5
	0x16000000 ～ 0x16FFFFFF	PCIメモリウィンドウ6
	0x17000000 ～ 0x17FFFFFF	PCIメモリウィンドウ7

おきます．CPUからのアドレスバスやリード/ライト方向，バイトイネーブル情報はこのprocess文の外で代入しておきます．

また，現時点ではテストレジスタしか用意していないので，バスアクセススタート信号をそのままテストレジスタのアクセススタート信号に代入しています．

テストレジスタ制御部では，バスアクセススタート信号がセットされれば，テストレジスタへのアクセスが開始されたと判断し，内部のシーケンサを動作させます．ライトアクセス時であれば，アドレスとバイトイネーブル情報にしたがってテストレジスタにデータを書き込みます．リードアクセス時であれば，アドレス情報にしたがってテストレジスタを選択し，その内容をリードバッファに出力します．なお，リードアクセス時はバイトイネーブル情報を無視して，バス幅全体(ここでは32ビット)のデータを用意してリードバッファに出力しています．読み書き動作が終了したら，アクセスレディ信号を返します．

ローカルバス制御部では，バスアクセススタート信号を1クロック期間だけセットした後，アクセスレディ信号がセットされるのを待ちます．アクセスレディ信号がセットされたら，リードアクセス時はリードバッファの内容をローカルデータバスの出力バッファに転送します．これでテストレジスタへの読み書きが完了したので，RDY信号をセットします．そしてリード/ライト信号がどちらもディアサートされるまで待ってから，アイドルステートに戻ります．

● バス制御レジスタの初期化例

リスト2にバス制御レジスタの初期化例を示します．リスト2では，スタティックメモリエリア1を，可変レイテンシI/O，32ビット幅，DRF = 7クロック，DRN = 7クロックに設定している例です．またRDY信号とnPWE信号を有効にするために，ファンクション設定レジスタと入出力方向レジスタも初期化しています．

4 PCIバスブリッジの実装

● FPGA内にPCIブリッジを実装

FPGA内部に実装したレジスタを読み書きできれば，あとはFPGA内にPCIブリッジを実装するだけです．ここでは，参考文献2)～5)で紹介されている，SH-4用に設計したPCIブリッジを移植しました．

PCIブリッジを実現するためには，PCIのメモリ空間とPCIのI/O空間をそれぞれPXA25xのメモリ空間の中にマッピングする機能が必要になります．PCIのメモリ空間は4Gバイトありますが，これはPXA25xの全メモリ空間を割り当てても足りません．そこで，数十Mバイト程度のメモリウィンドウを用意し，その中に4GバイトのPCIメモリ空間をバンク切り替えしてアクセスできるようなしくみを用意します．このメモリウィンドウをPCIメモリウィンドウと呼ぶことにします．今回は一つのPCIメモリウィンドウのサイズを16Mバイトとしました．

PCIのI/O空間もアドレッシング能力的には4Gバイトのサイズがありますが，x86系CPUのI/O空間が64Kバイトということもあり，ほとんどのPCIデバイスが下位の64Kバイトの範囲内で動作するようになっているので，PCIのI/O空間はバンク切り替えせずにリニアに64Kバイトをアクセスできるようにします．

また，CPUからPCIバス上に割り込みを出力したり，逆にPCIバス側からCPUに対して割り込みをかけられるようになると便利かと思います．

よって，PCIのメモリ空間やI/O空間以外に，PCIバスに関連した制御レジスタをどこかに用意しなければなりません．

〔リスト3〕PCIブリッジのVHDLソース(一部)

```
-- ***** PCIコントローラ定義 ***** --
PCIIF : PCI_CTRL
    generic map(
         BASEADDRESS    => 2 -- ベースアドレスレジスタ 2本
    )
    port map(

    -- PCIバス信号ピン

                            ～中略～

    -- 内部バス信号
         CLK                  => CLK,
         RST                  => RST,

    -- ターゲットデバイス内部バス
         TGT_ChipSelect => TGT_ChipSelect,
         TGT_Address          => TGT_Address,
         TGT_ByteEnable => TGT_ByteEnable,
         TGT_ReadWrite  => TGT_ReadWrite,
         TGT_AccStart   => TGT_AccStart,
         TGT_ReadData   => TGT_ReadData,
         TGT_WriteData  => TGT_WriteData,
         TGT_AccReady   => TGT_AccReady,
         TGT_Interrupt  => CPU2PCI_IntSta and CPU2PCI_IntMsk,

    -- PCIコンフィグレーションレジスタ情報
         CfgCmd_Mst           => PCI_CfgCmd_Mst,
         CfgCmd_Mem           => PCI_CfgCmd_Mem,
    --   CfgCmd_Io       => PCI_CfgCmd_Io,
         CfgSta_MAbt          => PCI_CfgSta_MAbt,
         CfgSta_TAbt          => PCI_CfgSta_TAbt,
         CPU_MSTAbort_Clr=> PCI_MSTAbort_Clr,
         CPU_TGTAbort_Clr=> PCI_TGTAbort_Clr,

    -- イニシエータデバイス内部バス
         MST_MemSelect  => Hit_PCIMemory,
         MST_IoSelect   => Hit_PCIIo,
         MST_CfgSelect  => '0',
         -- ↑コンフィグレーションサイクル発行なし
         MST_ReadWrite  => CPU_ReadWrite,
         MST_Address          => PCI_Address,
         MST_ByteEnable => CPU_ByteEnable,
         MST_AccStart   => PCI_AccStart,
         MST_ReadData   => PCI_ReadData,
         MST_WriteData  => CPU_WriteData,
         MST_AccReady   => PCI_AccReady

    );

    -- ***** ターゲットデバイス リードデータバス/レディ処理 ***** --
    TGT_ReadData <= PCI2SDRAM_ReadData when (TGT_ChipSelect(0) = '1')
                                       else PCI2CTRL_ReadData;
    TGT_AccReady <= PCI2SDRAM_AccReady when (TGT_ChipSelect(0) = '1')
                                       else PCI2CTRL_AccReady;
    -- ベースアドレス0 SDRAM空間
    PCI2SDRAM_AccStart <= TGT_ChipSelect(0) and TGT_AccStart;
    -- ベースアドレス1 PCIバス側制御レジスタ空間
    PCI2CTRL_AccStart  <= TGT_ChipSelect(1) and TGT_AccStart;

-- ***** PXA25xローカルバス制御部定義 ***** --
CPU0 : PXA25X_LOCAL
    port map(
         -- PXA25xローカルバス信号 --
         CLK            => X_BSDCLK0,
         nRST      => X_nRST,
         nCS            => nCS,
         R_nW      => X_BR_nW,
         nOE            => X_nBOE,
         nPWE      => X_nBPWE,
         nBE            => X_nDQM,
         ADR            => X_BA,
         DATA      => X_BD,
         RDY            => X_BRDY,

         DBG            => CPU_DBG,

         -- システム内部信号 --
         Reset          => RST,        -- システムリセット
         Clock          => CLK,        -- システムクロック
         Address        => CPU_Address,
         ReadWrite => CPU_ReadWrite,
         ByteEnable     => CPU_ByteEnable,
         AccStart  => CPU_AccStart,
         WriteBuff => CPU_WriteData,
         ReadBuff  => CPU_ReadData,
         AccReady  => CPU_AccReady
    );

    -- チップセレクト
    nCS(0) <= '1';      -- エリア0はFlashROM
    nCS(1) <= X_nBCS1;  -- エリア1はSDRAM共有メモリ空間
    nCS(2) <= '1';      -- エリア2は使用中
    nCS(3) <= X_nBCS3;  -- エリア3はPCI制御レジスタ空間
    nCS(4) <= X_nBCS4;  -- エリア4はPCIメモリ空間
    nCS(5) <= X_nBCS5;  -- エリア5はPCIメモリ空間

    -- ***** アドレスデコーダ ***** --
    Hit_SDRAM    <= '1' when (nCS(1) = '0') else '0';
    Hit_CTRL     <= '1' when (nCS(3) = '0' and CPU_Address(25) = '0')
                        else '0';
    Hit_PCIIo    <= '1' when (nCS(3) = '0' and CPU_Address(25) = '1')
                        else '0';
    Hit_PCIMemory <= '1' when (nCS(4) = '0' or nCS(5) = '0')
                                                        else '0';

    -- ***** PCIバスイニシエータアドレス生成 ***** --
    PCI_Address(31 downto 24) <= (others => '0') when (nCS(3) = '0')
                                 ～中略～
    PCI_Address(23 downto  0) <= CPU_Address(23 downto  0);
                                 ～以下略～
```

● PCIブリッジのメモリマップ

以上を考慮したPCIブリッジのメモリマップを表3(p.153)に示します.

今回使用したFPGA評価キットであるStratixボードには,標準で8MバイトのSDRAMも実装されています.そこでエリア1にStratixボード上のSDRAMを割り当てました.また制御レジスタ空間はエリア3の下位アドレスに,PCIのI/O空間はエリア3の上位アドレスにマッピングしました.そしてエリア4とエリア5を16Mバイトずつ八つのPCIメモリウィンドウを配置します.それぞれのPCIメモリウィンドウのベースアドレスを16Mバイトずつずらしたアドレスに設定すると,最大128MバイトのPCIメモリ空間をリニアにマッピングできることになります.

リスト3にPCIブリッジのVHDLソースを示します.CPUのローカルバス部分には,リスト1(a)の設計をそのまま使います.

参考文献

1)『PCIデバイス設計入門』,TECH I Vol.3,CQ出版(株)
2) 山武一朗,「SH-4用PCIバスブリッジの設計/製作(バスマスタ基礎知識編)」,『Interface』,2000年10月号
3) 井倉将実,「SH-4用PCIバスブリッジの設計/製作(イニシエータ基礎知識編)」,『Interface』,2000年11月号
4) 来須川智久,「SH-4用PCIバスブリッジの設計/製作(PCIバスブリッジ設計編)」,『Interface』,2000年12月号
5) 菅原尚伸,「SH-4用PCIバスブリッジの設計/製作(応用システム構築事例編)」,『Interface』,2001年2月号

やまたけ・いちろう　来栖川電工(有)

仕様の記述からコードの自動生成まで実現可能な

プロトコル仕様記述言語CMDLによる通信プロトコル設計

日下志 友彦

はじめに

近年，企業，家庭，自動車など，あらゆる分野においてネットワークが急速に広がっています．ネットワークにつながれた機器は，おたがいが通信を行うための決め事である“通信プロトコル”によってデータをやりとりし，さまざまなサービスを実現します．現在，多数の通信プロトコルが提案されており，標準化も進められています．最近は実現するサービスが高度になり，これらの仕様も複雑化しています．

このような状況において，ネットワーク機器や組み込み機器の開発では，新しい通信プロトコルをいち早く取り入れて実装することが重要になってきます．これまでの通信プロトコル開発では，まず設計者が仕様書を理解し，そして実装するターゲットに応じた設計言語や設計環境を用いて，それぞれに開発する必要がありました．

こういった問題に対し，米国Novilit社は，通信プロトコルの仕様を専用言語を用いて記述することで，ターゲットに依存することなく統一的に通信プロトコルを設計できる手法を提案しています．この手法を用いれば，通信プロトコル設計が容易になり，また，設計期間を大幅に短縮することができます．本稿では，Novilit社の開発手法を採用したプロトコル設計環境であるAnyWareを用いた，まったく新しい方法による通信プロトコル開発を説明します．

通信プロトコルとは？

通信プロトコルとは，機器同士がデータをやりとりするために規定された約束事のことで，通信規約と訳されることもあります．もっとも有名なプロトコルモデルとして，ISO（International Organization for Standardization：国際標準化機構）が規定したOSI基本参照モデルがあります（図1）．このモデルでは，通信に必要な規約を，物理層からアプリケーションレベルまで七つの層に区分しています．

物理層では，通信ケーブルや電気信号に関する規約が規定されていますが，第2層より上位の層では，データのフォーマットとデータを受け取ったときの動作の二つを中心に規定されます．

AnyWareでは，データフォーマットの規定される第2層より上位を対象とし，物理層レベルとは，ライブラリを通じてインターフェースをとることができるようになっています．

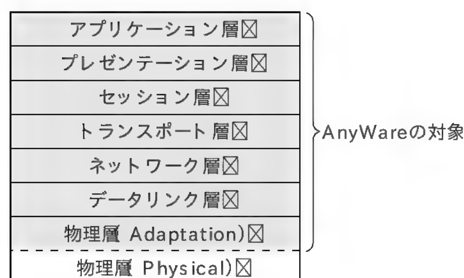
AnyWareとは

前述したとおり，AnyWareは，Novilit社が開発した通信プロトコル開発環境です（図2）．AnyWareでは，Novilit社が開発したプロトコル仕様記述言語であるCMDL（Communication Machine Definition Language）を使って設計することができます．この言語は，通信プロトコルの仕様記述に特化した言語になっており，あらゆる通信プロトコルの仕様を簡単に記述することができます．また，高度なプロトコル合成エンジンであるCMDE（Communication Machine Development Engine）を用いることで，CMDLで書かれたプロトコル仕様をコンパイルし，各種ターゲットに実装可能なソフトウェアやハードウェアを自動的に生成することができます．

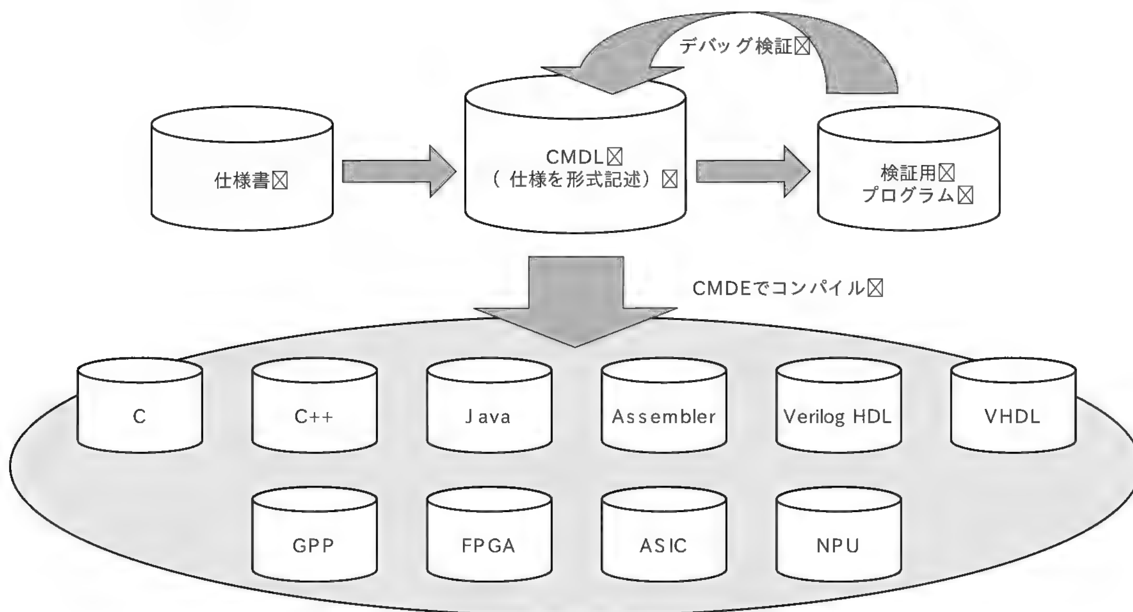
また，あらゆるプロセッサ環境で動作するプロトコルを開発することができ，使用するOSにも依存しません．そのため，ポータブルな通信プロトコルスタックを開発することができます．AnyWareは，FPGAやDSP，NPU，ASIC，汎用プロセッサなどに実装可能なソフトウェア，ファームウェア，ハードウェアを設計することができます．たとえばソフトウェア向けにはC/C++を，ハードウェア向けにはVerilog HDLやVHDLなどのコードを結果として出力することができます．

さらには，AnyWareは実用的なプロトコルスタックを生成するために必要なツールを含めた完全な開発環境で，CMDLと

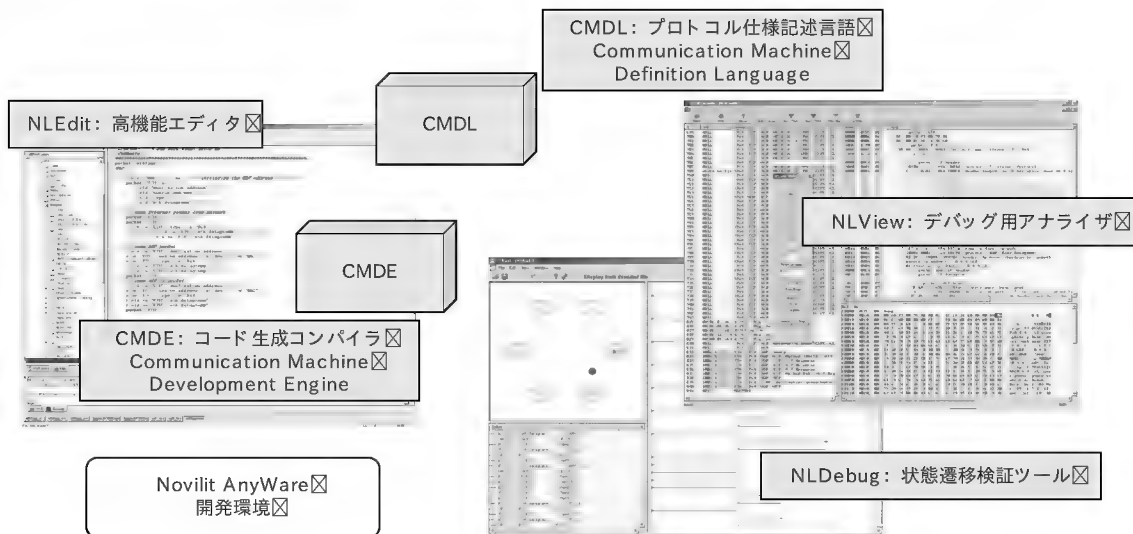
〔図1〕OSI基本参照モデル



〔図2〕AnyWareによるプロトコル設計



〔図3〕AnyWareの構成



CMDEに加え，専用のエディタ（NLEdit），検証用のネットワークエミュレータ（NLDebug），プロトコルアナライザ（NLView），およびターゲット環境のためのライブラリを含んでいます（図3）．

AnyWareでは，CMDLを用いた仕様レベルでの設計になるので，仕様をターゲットの環境に合わせて，どうコーディングするかを考える必要がありません．設計者は，英語や日本語などの自然言語で書かれた仕様書を，形式的言語であるCMDLで置き換えるというレベルで設計することができます．また，CMDLで記述されるコードは，仕様レベルなので，可読性が非常に高くなります．したがって，プロトコルの特徴といってもよい仕様の変更や追加にも即座に対応することが可能になります．

また，コーディング量が少なくても済むことに加え，検証やデバッグの繰り返しが減少するため，通信プロトコルの開発期間を大幅に短縮することができます．

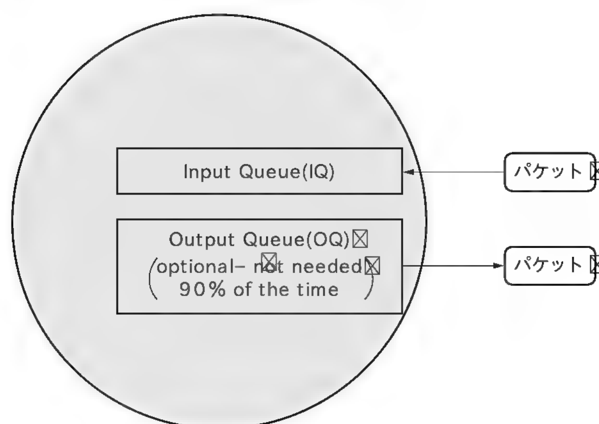
AnyWareの技術概要と開発フロー

本項では，AnyWareの技術概要とAnyWareを用いた場合の開発のフローについて説明します．

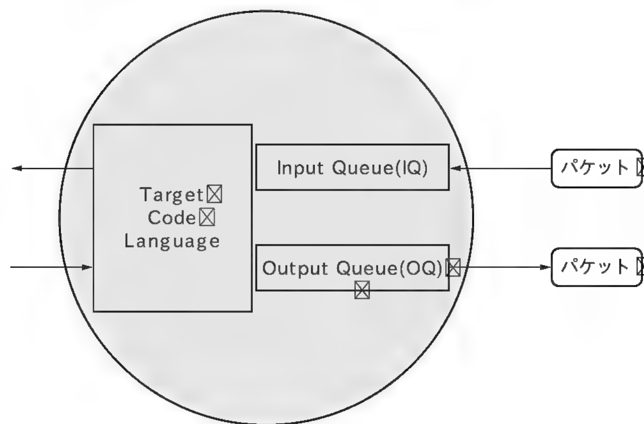
● コミュニケーションマシンとプロトコル

ここでは，一般的に通信を行うオブジェクトのことを，コミュニケーションマシンと定義します．コミュニケーションマ

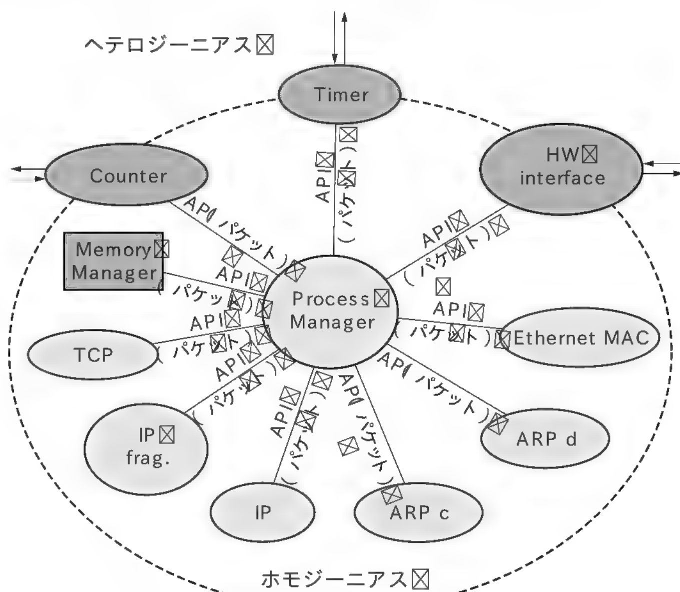
〔図4〕ホモジーニアスコミュニケーションマシン



〔図5〕ヘテロジーニアスコミュニケーションマシン



〔図6〕分割されたコミュニケーションマシン(TCP/IPでの例)



シンは，他のコミュニケーションマシンからのメッセージを受け取って制御されたり，データを得ることができます．また，逆に他のコミュニケーションマシンにメッセージを送ることで反応を返すことができます．

通常，コミュニケーションマシンは，OSIの階層モデルや実装をハードウェアにするのかソフトウェアにするのかといった違いにより，機能を分割(パーティショニング)します．しかし，従来のこれらの分割方法は，単なるモデルでしかないので，分割された部分間のやりとりが複雑になります．そこで，AnyWareでは，コミュニケーションマシンをホモジーニアス(Homogeneous)とヘテロジーニアス(Heterogeneous)と呼ぶ二つのタイプのコミュニケーションマシンに分けるという独自の分割アルゴリズムを用います．

ホモジーニアスコミュニケーションマシンとは，パケットによるインターフェースのみで他のコミュニケーションマシンとやりとりを行うもので，パケットの入力に対し，処理を行ってパケットの出力を行うという単純な動作を行います(図4)．

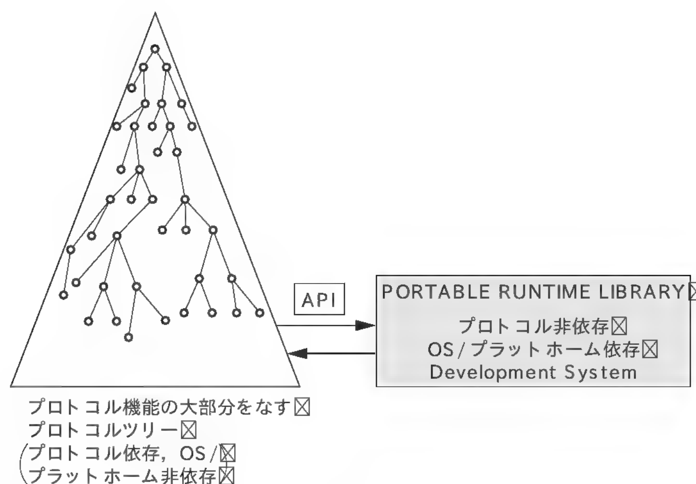
これに対して，ヘテロジーニアスコミュニケーションマシンは，パケットに加え，外部とのインターフェースが必要なコミュニケーションマシンです(図5)．ここでいう外部とのインターフェースは，OSのシステムコールや物理的なケーブルのインターフェースなどのことで，ターゲットシステムを記述している言語(C，C++，VHDLなど)で記述されることになります．

このように二つのタイプに分けられたコミュニケーションマシンの例の一つとして，TCP/IPでの例を図6に紹介します．この例のように，全体を管理して制御するプロセスマネージャ(Process Manager)が，パケットによる単純なインターフェースでそれぞれのコミュニケーションマシンと通信を行い，また，ヘテロジーニアスコミュニケーションマシンを通じて，タイマやカウンタ，ハードウェアなどのターゲットに依存する外部と通信を実現しています．このように分割することで，コミュニケーションマシンを非常に簡単な機能パートごとに分割することができ，また，ターゲットに依存する部分をヘテロジーニアスコミュニケーションマシンとして，切り分けて考えることができます．

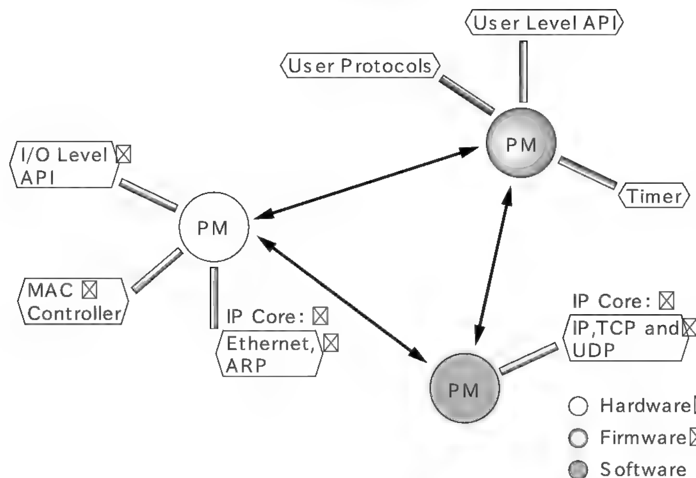
AnyWareで通信プロトコルを設計する場合，図7にあるように，プロトコルを構成する大部分であるホモジーニアスコミュニケーションマシンをCMDLによってツリー構造で定義し，外部とのインターフェースなど，ターゲットに依存する部分については，ライブラリAPIを用いるという構成になります．OSなどのターゲットに依存する部分は，プロトコルツリーとは完全に切り離され，最小限の大きさになっているので，ターゲットに応じたライブラリを使い分けることで，各種ターゲットに対応できるポータビリティを持った設計ができるようになります．

また，分割されたコミュニケーションマシンは，ハードウェア，ファームウェア，ソフトウェアのいずれかに実装すること

〔図7〕プロトコルアーキテクチャ



〔図8〕ハードウェア/ソフトウェアの協調設計



ができ，おたがいに非同期にやりとりすることが可能です(図8).

分割されたコミュニケーションマシンは，次の三つのパートの機能に分けて考えることができます.

- ビットストリームからメッセージへデコードするビットストリームパーサ
- 内部の機器のふるまいを定義するステートマシン
- メッセージからビットストリームをエンコードするビットストリームジェネレータ

CMDLはこのような観点から開発されており，通信プロトコルとコミュニケーションマシンのすべての部分を形式的に開発することを可能にします．AnyWareでは，いくつかの階層，パートに分けられた広義のコミュニケーションマシン，および通信プロトコルをCMDLで記述します.

CMDLは宣言的言語で，Cのような手続き的言語ではありません．プログラムのフローというよりは，データに即したイベントを記述していくことになります．また，オブジェクト指向で，決定論的言語であることから，実装において曖昧(あいまい)性を含まない論理ツリーを記述することができます.

● AnyWareによるプロトコルの設計フロー

通信プロトコルの設計フローは，図2にも示したとおり，次のようになります.

1) プロトコル仕様をCMDLで記述
2) デバッグ用プログラムを作成
3) NLDebug，NLViewを用いて仕様レベルで検証・デバッグ
4) CMDEによりターゲットコードを出力

CMDLでのプロトコル仕様記述

● CMDLの特徴

プロトコルの仕様を記述するために開発されたCMDLには，次のような特徴があります.

- 通信プロトコルに特化した仕様記述言語
- あらゆる通信プロトコルを記述可能
- キーワードが少なく学習が容易
- オブジェクト指向
- 宣言的(非手続き的)
- インラインコード制御

CMDLはキーワードが28しかないため，数日あれば学習できます．また，オブジェクト指向言語なので，機能やデータをオブジェクトとして捉えて設計することができます．さらに，CMDLのコード内にC/C++やHDLのコードを組み込むことができるインライン制御機能をもっており，拡張性も備えています.

CMDLでは，通信プロトコルで使用するデータ(パケット，フレーム)の定義，およびステートマシンで記述された機器のふるまいを記述します．また，外部インターフェースとの接続やライブラリで定義されたタイマなどの割り込みを定義することもでき，検証のためにキーボード入力に対応したテストデータの送信を定義することなどができます.

● CMDLの基礎

では，CMDLの記述例を簡単に紹介します．通常，通信プロトコルで用いるストリームデータは，ヘッダ部分とデータ部分が階層的につながった構成のデータを用います．通信プロトコルで用いるこのデータフォーマットのことをパケット，もしくはフレームと呼び，それぞれのデータ要素をフィールドと呼びますが，CMDLでは，packetというキーワードとfieldというキーワードを用いてデータ構成の定義を行います.

もっとも簡単な例として，8ビット長の一つのフィールドを持つパケット(図9)をCMDLで記述した場合，次のようになります.

```
packet("a",8)
```

packetというキーワードの後に続く括弧で囲まれる中に，そのパケットの属性を記述します．この例では，""で囲まれた

〔表1〕属性の指定

フル表示	簡略表示	定　義
init	i	初期値の設定
len	l	ビット長の指定
lenregexpr	lre, lere, lenre, lregexp	正規表現によるパターンマッチングを用いたビット長の指定
ma	m	メモリ領域の指定
repeat	r	連続するパケットの指定
send	s	パケットの送信

文字列が，そのパケットを識別する名前となり，数字はビット長です．識別子として用いられる文字列には，大文字小文字を区別したすべてのアルファベットとスペースを用いることができ，たとえば仕様書の中で使われているキーワードをそのまま識別子として使用することができます．また，ビット長をより明示的に表現するためには，

```
packet("a",len=8)
```

というように，lenというキーワードを用いて属性を指定することができます．lenの他にも初期値や利用するメモリ領域などを指定するためなどにさまざまな属性を指定するためのキーワードが準備されています(表1)．

次に，複数のフィールドを持つ例を示します．図10にあるように，それぞれ，3ビット長，4ビット長，1ビット長のa，b，cという名前のフィールドを持つパケットを記述する例です．なお，C言語と同様に，//〜と/* 〜 */を用いてコメントを記述できます．

```
packet("a") {        // (1)
  field("a",3)       // (2)
  field("b",4)
  field("c",1)
}
```

このように，packet("a")という記述に続いて，fieldというキーワードを用いてそれぞれのフィールドを定義します．この例では，packetに続く括弧の中に，パケット長に関する記述がありませんが，続いて定義されるフィールドに応じて自動的に計算されます

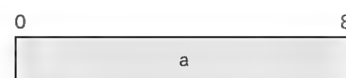
また，(1)，(2)で，"a"という名前が二つ用いられていますが，どちらもローカルな名前として区別して解釈されます．

プロトコルの仕様書では，パケットが階層的に定義されることも少なくありません．次の例では，図11のように階層的に定義されるパケットをCMDLで記述しています．defというキーワードを用いて，階層的に定義します．

```
packet("a") {
  packet("a")     // (1)
  field("b",16)
  packet("c")     // (2)

  def:
    packet("a") {      // (1) のパケット"a"を定義
      field("a",11)
      field("b",14)
      field("c",7)
    }
    packet("c") {      // (2) のパケット"c"を定義
      field(“ a” ,8)
      field(“ b” ,8)
    }
}
```

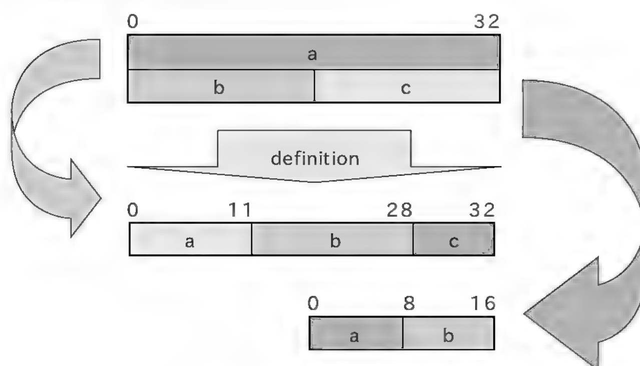
〔図9〕一つのフィールドからなるパケット



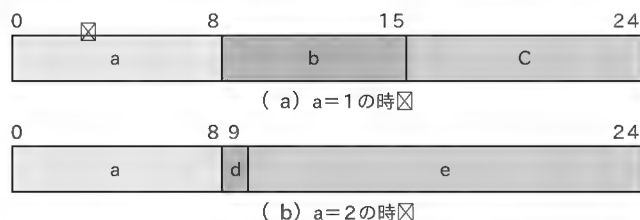
〔図10〕複数のフィールドからなるパケット



〔図11〕階層的に定義されるパケット



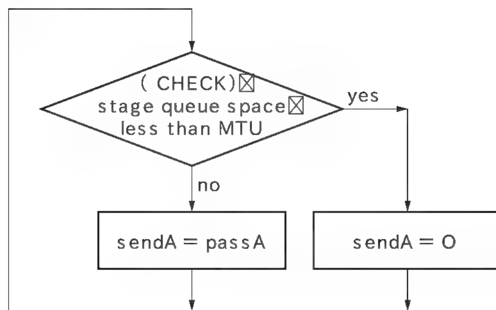
〔図12〕分岐した定義のあるパケット



特定のフィールドの値などの条件により，パケットが分岐構造を持って定義されることもあります．たとえば図12の例では，"a"の値が1の場合は7ビット長の"b"と9ビット長の"c"というフィールドが続いて定義され，"a"の値が2の場合は，1ビット長の"d"と15ビット長の"e"というフィールドが定義されます．このようなパケットは，altという分岐を表わすキーワードを用いて，次のようにCMDLで記述されます．

```
packet("a") {
  field("a",8)
```

〔図13〕ステートマシン



〔表2〕ステートテーブル

Current state		Row	Next state	
state	condition		action	state
CHECK	depthStageBuffor <(sizeStageBuffer-MTU)	1	sendA=0	CHECK
	−	2	sendA=passA	

〔リスト1〕CMDLの記述例

```
alt(v("state")) {      /* "state" の値を判定  */
   [1]        /* "state" が1のとき */
    /* "depthStageBuffer"の値が、"sizeStageBuffer"と
        "MIU"の差より小さいかどうかを判定 (A) */
    alt(v("depthStageBuffer")<(v("sizeStageBuffer")-v("MIU"))) {
        [1]                                  /* Aの条件がTrueのとき   */
            v("sendA") =0                    /* "sendA"の値を0に */
            packet("send",s=pid(^<"server")) /* パケット"send"を"server"に送信 */
            v("state")=1                     /* "state"を1に */
        [0]                                  /* Aの条件がFalseのとき  */
            v("sendA")=v("passA")            /* "sendA"に"passA"の値を代入 */
            packet("send",s=pid(^<"server")) /* パケット"send"を"server"に送信  */
            v("state")=1                     /* "state"を1に */
      }
}
```

```
    alt(v("a")) {
        [1]field("b",7)field("c",9)
        [2]field("d",1)field("e",15)
    }
  }
```

altに続く括弧の中に条件を判定する値や式が記述され，[ ]に囲まれる中に判定される値を記述します．なお，v("a")という記述は，フィールド"a"の値を参照することを意味します．

● CMDLの構成

CMDLにより通信プロトコルを記述する際には，大きく次の三つのセクションに分けて記述されます．

1) definitionセクション(def)
2) variableセクション(var)
3) processセクション(pro)

definitionセクションでは，前項で説明したようなパケット構造の定義がされます．def:というキーワードに続き，パケットの定義を行います．

variableセクションはオプションパートで，定数や変数として定義されるpacket，もしくはfieldを定義します．状態変数などの宣言に用いられ，var:というキーワードに続いて，次のように記述されます．

```
var: field("state",8)
```

processセクションでは，プロトコルの処理を記述します．たとえば，IPアドレスを他の機器に送信するといったような処理が記述されます．このパートでは，初期化(ini)，デコード(dec)，エンコード(enc)のセクションに分けて記述することができます．ini{ }で囲まれる部分は初期化の際に処理され，dec{ }，enc{ }で囲まれる部分は，それぞれデコード，エンコードの際の処理されます．dec，encのどちらにも囲まれていない部分は，デコード，エンコードの両方のときに処理されます．よって，CMDLで記述ファイルは，次のような構造になります．

```
packet("<パケット名>",<属性>) {
  var: ...        // 変数の宣言
  pro: ...// プロトコル処理の記述
      ini{     // 初期化の処理
         ...
       }
         // デコード/エンコードの処理
         ...
      dec{     // デコード処理
         ...
       }
      enc{     // エンコード処理
         ...
       }
  def:     // パケット構造の定義
        ....
}
```

● ステートマシンの記述

通信プロトコルのふるまいは，ステートマシンで表現することができます．また，このステートマシンは，ステートテーブルに変換することができます．たとえば，図13のステートマシンは，表2のステートテーブルに変換することができます．CMDLでは，このステートテーブルを言語に置き換えて記述します．表2をCMDLで表現したものをリスト1に示します．

この例にあるように，sendというキーワードを属性として指定することでパケットの送信を記述することができます．

● 簡単なサーバプログラムの例

CMDLの記述例として，非常に単純化したサーバのプログラムの例をリスト2に示します．この例では，4ビットのヘッダ部と32ビットのデータ部からなるパケットを用いて通信を行います．サーバ側ではアクティブ状態のときに，クライアントか

〔リスト2〕簡単なサーバの記述例

```
packet("Server") {

  var:  field("state",4)    /* 状態変数"state"の宣言 */
        packet("reply")        /* クライアントに送信する
                                  パケット"reply"の宣言 */

  pro:
    ini {
       /* 状態変数"state"を0に初期化 */
       field("state",4,ma=1,i=0)
    }

    dec {
      alt(sn) {
        [pid(^<"Client")]            /* Clientからのイベント */
           packet("request")        /* パケット"request"として
                                       データを受信 */
           alt(v("state")) {        /* 状態変数"state"を判定
                                       (1の時のみ動作) */
              /* パケット"request"の"header"フィールドを判定 */
              [1] alt(v("request">"header")) {
                   /* "request"の"header"が1のとき,
                      "header"が3, "data"が1234の
                      パケット"reply"をClientに送信 */
                    [1] v("reply">"header")=3
                        v("reply">"data")=1234
                        packet("reply",send=pid(^<"Client"))
                   /* "request"の"header"が2のとき,
                      "header"が4, "data"が2345の
                      パケット"reply"をClientに送信 */
                    [2] v("reply">"header")=4
                        v("reply">"data")=2345
                        packet("reply",send=pid(^<"Client"))
                  }
              [0]  0   /* "state"が0のときは何もしない */
           }
      }
    }

  def:
     packet("reply") {       /* パケット"reply"の定義 */
       field("header",4)
       field("data",32)
     }
     packet("request") {     /* パケット"request"の定義 */
       field("header",4)
       field("data",32)
     }
}
```

らヘッダ部が1もしくは2のパケットを受信した場合に，ヘッダ部が3で，データが1234もしくは2345のパケットをクライアントに送信します．なお，この例では，簡単にするため異常なデータを受け取った場合の処理などは省略しています．

● TCPの例

TCPのヘッダをCMDLで定義した例の一部をリスト3に示します．TCPの仕様で定められるパケットの構造がCMDLで簡単に記述できることがわかると思います．この例では，パケットの定義のみですが，このレベルの記述だけで，エンコード，およびデコードの処理部分のプログラムが実現されます．また，ステートマシンを記述することで，プロトコルスタックとして動作します．

● NLEditとコンパイル

CMDLは，汎用のエディタを用いても記述できますが，AnyWareでは，CMDLによるプロトコル設計をサポートするNLEdit（図14）というエディタを提供しています．予約語の色付け機能や，括弧の対応などのチェック機能があります．また，キーワードをコピーするだけで，そのキーワードを識別子としたパケットやフィールドの雛型をペーストするような入力支援機能もあります．

また，エディタとしての機能の他に，プロジェクトの管理機能や直接CMDEコンパイラを起動する機能も備えています．たとえば，GUI画面上にコンパイル処理のスクリプトに対応する

〔リスト3〕TCPのヘッダをCMDLで定義した例（一部）

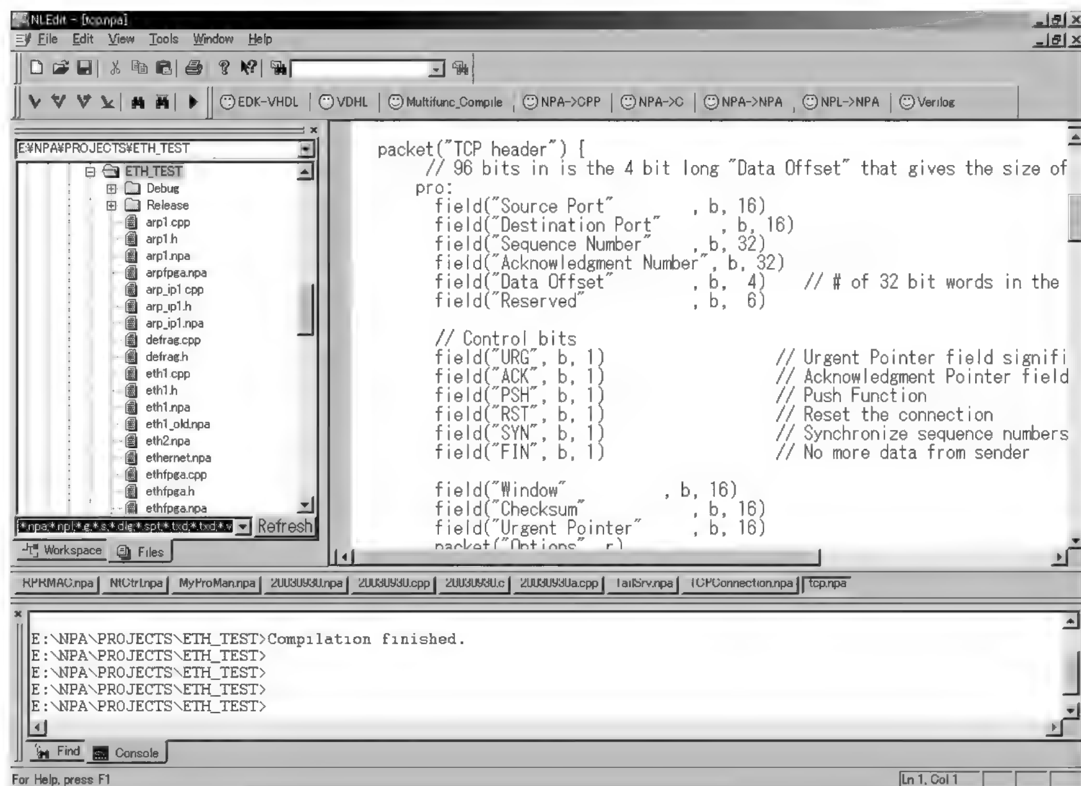
```
packet("TCP header") {
     // 96 bits in is the 4 bit long "Data Offset"
     // that gives the size of the header.
     pro:
       field("Source Port"           , b, 16)
       field("Destination Port"      , b, 16)
       field("Sequence Number"       , b, 32)
       field("Acknowledgment Number", b, 32)
       field("Data Offset"           , b,  4)
            // # of 32 bit words in the TCP header
       field("Reserved"              , b,  6)

       // Control bits
       field("URG", b, 1)   // Urgent Pointer field significant
       field("ACK", b, 1)   // Acknowledgment Pointer field
                            //  significant
       field("PSH", b, 1)   // Push Function
       field("RST", b, 1)   // Reset the connection
       field("SYN", b, 1)   // Synchronize sequence numbers
       field("FIN", b, 1)   // No more data from sender

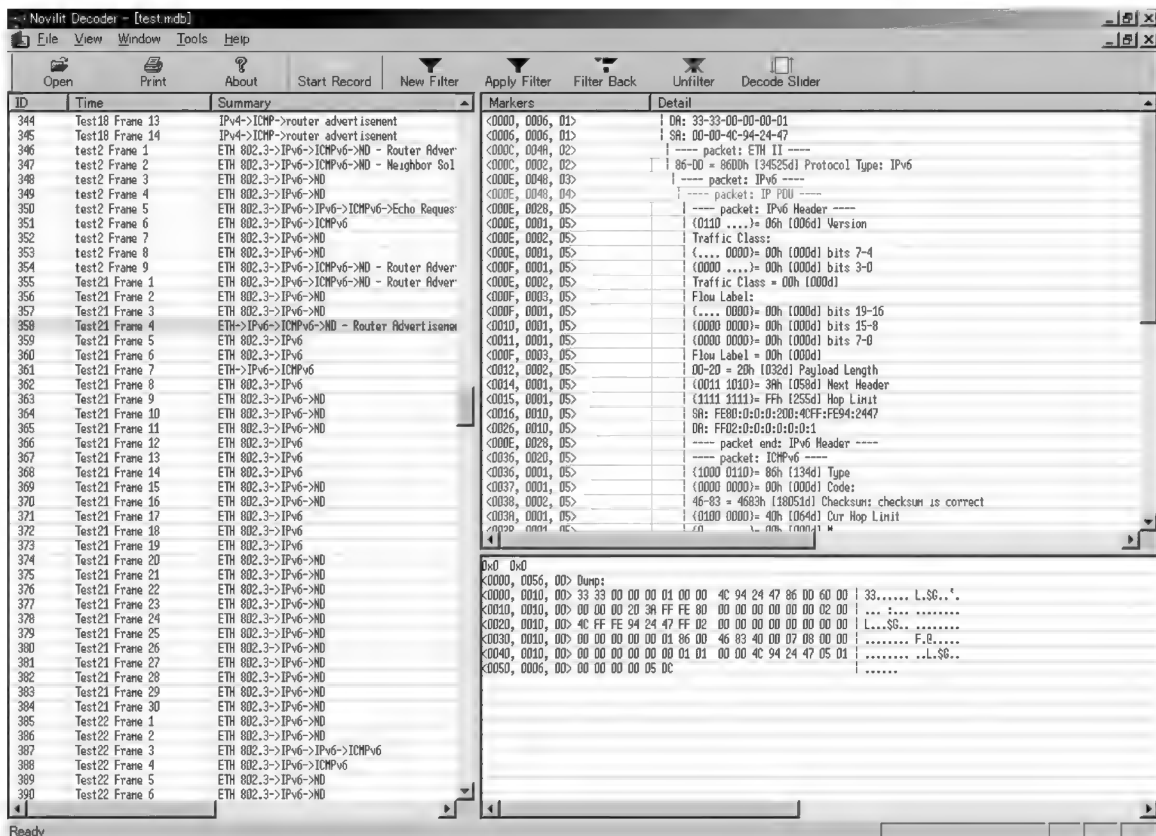
       field("Window"                , b, 16)
       field("Checksum"              , b, 16)
       field("Urgent Pointer"        , b, 16)
       packet("Options", r)
       f("padding", b, 8,r=((((cio + 3)>>2)<< 2) - cio))
     def:

     packet("Options") {
         var:
               p(tn=^<"NtCtrl"<"Message", on="msg")
         pro:
               f("option kind", 8)
               // Branch on the option kind
               alt(v("option kind")) {
                [0] break /* packet("End option")   */
                [1] 0 // packet("Nop option")
                [2] packet("Max segment size option")
                [8] packet("Timestamp option")
                // size of field is 8 bits,
                // so max value can be 0xFF
                [3..7.9..0xFF]
                     packet("Undefined option", la(0,8)*8)
               } // end alt(la(0,8))
         def:
            packet("Max segment size option", 32) {
            var:
               p(tn=^<"NtCtrl"<"Message", on="msg")
            pro:
                field("length", 8) // length=4
                alt(lv) {
                 [4] 0  // expected size
                 [0..3.5..0xFF]
                 // send message to NtCtrl to inform
                 // user of problem "Was not 4 !"
                     dec {
                         packet("msg",i)
                         field("msg"<"Message Index", i= 1)
                         field("msg"<"Message Value",
                               i= v("length"))
                         packet("msg", s=pid(^<"NtCtrl"))
                     }
                }
                field("Max segment size", 16)
            }
```

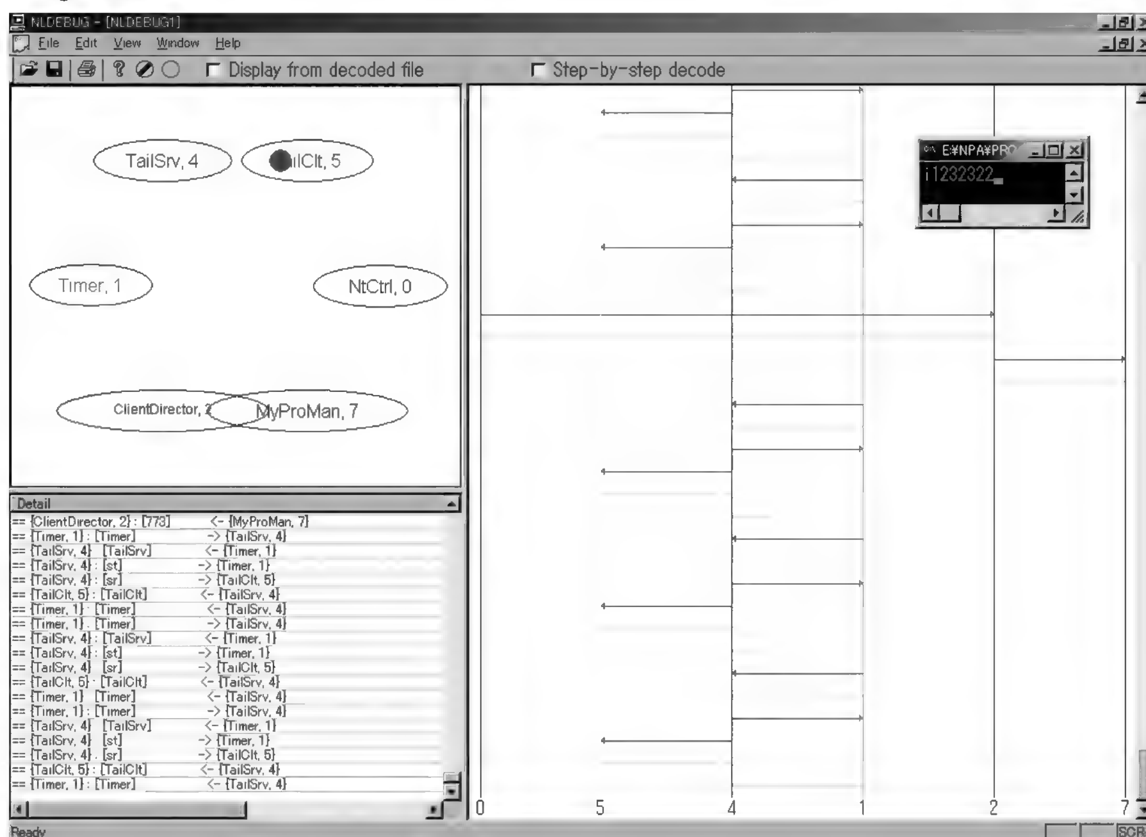
〔図14〕NLEditの画面



〔図15〕NLViewの画面



〔図16〕NLDebugの画面



CやC++，VHDL，Verilog HDLなどのボタンがあり，設計者は，これらをクリックするだけで，開いているCMDLファイルをコンパイルすることができます．

検　証

AnyWareでは，NLView，NLDebugという二つのツールを用いて，仕様レベルでの検証を行うことができます．

● NLView

NLView（図15）は，プロトコルアナライザで，ネットワーク上に流れるパケットをキャプチャし，デコードして表示します．新たにCMDLで記述されたプロトコルのプロトコルアナライザとして使用することができ，これにより，デコードやエンコードが正しく行われているかどうかを確認することができます．

● NLDebug

NLDebug（図16）は，プロセッサ間でのデータのやりとりを可視化し，表示することができる検証ツールです．CMDLでは，あらかじめキーボード入力に対応して入力するテストデータを定義しておくことができます．NLDebugでは，このCMDLで動作が定義されたキーをインタラクティブに入力することで，コミュニケーションマシン間のデータの流れや通信プロトコルの動作が正しいかどうかをシミュレーションし，確認することができます．

また，キーボード入力の他にも，実際のネットワークからのデータを用いて検証することも，あらかじめ実行したデータを保存しておき，ワンステップごとに動作を確認することもできるようになっています．

まとめ

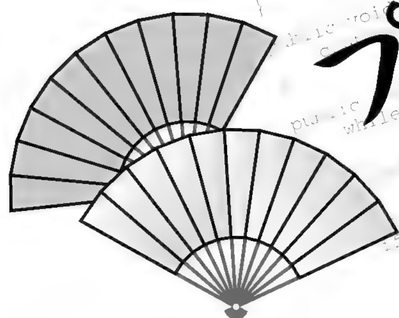
通信プロトコルの仕様記述に特化したCMDLを用いて，実装するターゲットに依存せずに仕様レベルで通信プロトコルを設計する手法について説明しました．多様で複雑化するプロトコル開発におけるブレークスルーとなり，プロトコル設計を効率化し，また品質を高める新しい技術になると考えられます．なお，AnyWareは，サンプルコードを含め，すべての機能を試すことができる21日間の無料評価が可能です．興味がありましたら，ぜひ一度，実際にツールを操作し，CMDLによるプロトコル開発を体感してください．

■AnyWareの問い合わせ先
NTTアドバンステクノロジCADシステム事業部
E-mail： info@cad.ntt-at.co.jp

ひがし・ともひこ　NTTアドバンステクノロジ（株）



プログラミングの要



宮坂 電人

第9回 ピーターの法則，そしてパターンとアルゴリズム

ピーターの法則

1970年代ごろ，『ピーターの法則』というビジネス関係の書籍が話題になったことがありました．これは，

- 階層社会においては，構成員は各自の器量に応じて，それぞれの無能のレベルに達する

というものです．平社員のときに優秀な人が出世して階級が上がっていくが，上がれば上がるほどだんだん能力が落ち，無能になった時点のポストでそのまま留まるという皮肉なことが書かれていたそうです．残念ながら筆者は同書を読んだことがないので本当にそんなことが書かれていたのかどうか確認できていません．そういえば日本では，

- 生まれたときは天童，幼少のころは天才，学生になると秀才，二十過ぎればただの人

という意味の揶揄(やゆ)があります．いずれも共通するのは本人の成長とまわりの期待に相対的なギャップがあり，成長するにつれてギャップが広がり，だんだん評価が落ちたというのがその正体ではないかと想像します．そして同様の事情が，ソフトウェア開発でも起こり得ます．

- 学生時代は天才プログラマ，入社当時は優秀な新人プログラマ，10年たてば普通のサラリーマン，35歳で無能(?!)になり引退

という図式に当てはまる人はけっこういるかもしれません(?!)．注意してほしいのは「天才プログラマ」と自称していた時期の事情と「無能になり引退」と評価される時期の事情があまりにも違いすぎて，同じ土俵で比較や評価ができないということです．学生時代はプログラミングのスケールが個人的な規模であったのに対し，「引退」直前の時点ではチーム化したり大規模化しているからです．チーム化したり大規模化したものが，学生時代の器量で挑戦して太刀打ちできるとは思えません．本人は成長しているはずなのですが，その成長のスケールとまわりから期待される成長のスケールにギャップが生じているわけです．

開発には規模がある

開発の規模に着目すると，

① 個人レベル：個人の裁量で開発が制御できる．プログラムは比較的，小規模．全ソースを自分の能力で把握できる
② チームレベル：3人から数人程度で開発を行う．プログラムは中規模．全ソースを自分の能力で把握できなくもないが苦しい
③ 企業レベル：複数の企業からなるチームで開発を行う．プログラムは大規模．全ソースを自分の能力で把握するのは不可能

といった三つの段階があると思われます．この分類でさきほどの「天才プログラマ」氏はおそらく，①個人レベル：学生時代は天才プログラマ，②チームレベル：入社当時は優秀な新人プログラマ，③企業レベル：普通のサラリーマン，35歳で無能になり引退という境遇だったのではないかと想像します．さて，冗談はここまでにしておき，それぞれのレベルで遭遇するであろう状況を細かく洞察していきましょう．

- 個人レベル：開発の主体はほとんど自分だけで制御できるので，誰かとの意思疎通はほとんど不要．取り扱う規模は小さいが，油断するとごちゃごちゃになるので自分で何らかの「整理」が必要になる．プログラムのマクロレベルとミクロレベルは区別しにくいし，区別すべき理由も見い出しにくい．
- チームレベル：開発の主体はチームであり，自分の独断では進めにくい．しかし全体の中での自分の位置付けを向上させることで(露骨にいえば権力を握ったり出世するといった意味)，ある程度は制御がきく．取り扱う規模は小さくないので個人レベルのときと同じ感覚ではそろそろやりにくくなる．チーム同士の意思疎通をはかるための「図面(アーキテクチャレベルでの構造を見わたせる図面．UMLなどのことをいっている．フローチャートのようなソースを読めばわかるようなミクロレベルのものではない)」が必要になる．プログラムのマクロレベルとミクロレベルの区別をつけないと，まずい状況が出はじめる．
- 企業レベル：開発の主体は「顧客」にあるが，顧客は必ずしも開発のプロではないので，とんでもない要求にふりまわされたり無駄足を踏むことも多い．もちろん自分の制御はすでに効いていない．取り扱う規模は巨大であり，複数企業にまたがって意思疎通をはかるための「文書」を大量に要求されるため，文書作りに労力を費やして肝心の開発がおろそかになる場合もあ

Column1

暗黙知と形式知

暗黙知と形式知というのを初めて聞く読者もいるかと思うので，少し解説しておきます．この言葉はハンガリーの物理学者マイケル・ポランニーによって提唱されたもので，簡単にいえば，

- 暗黙知（tacit knowledge）：明確に文章や形式として示すことができない個人的な体験に根ざす知識
- 形式知（explicit knowledge）：明確に文章や形式として示すことができ他人に伝達しやすい知識

ということです．この説明で気づいた人もいるでしょうが，いわゆる「職人の勘」とか「長年の経験」と称しているのが暗黙知に相当するもので，それに対し，学校や新人研修などで教えられマニュアル化が簡単なものが形式知に相当するものです．

形式知は講座や研修などで教えたり書籍化されることが多いので，これはとくにどう伝達すべきかで悩むことは少ないのですが，問題は暗黙知です．これを教えるために企業ではよくOJTと称した実地研修を利用するのですが，注意しないといけないのは，個人的な体験として習得させるので個人差が大きく，間違って自分に都合のいいように知識を受けとめたり構築されるおそれがある点です．「うちはOJTで実践的にやっている」と自慢している企業はいくつかあります．しかし，よく観察すると，完全に一人一人が自分勝手な思い込みで動いていて，そうでもないという実例があります．つまり，研修させているつもりが単なる「放任」であったり「放し飼い」にすぎない危険があるということです．

そうならないように管理者がきちんと正しい方向に誘導する手間ヒマをかけることが大事です．どうも企業によっては手間ヒマをかけたくないからOJTを採用しているように思えてなりません．また，なるべく暗黙知を積極的に形式知に転換させていく工夫が大事でしょう．暗黙知だけに頼っているのは個人レベルならまだ許されるかもしれませんが，企業レベルやチームレベルでは疑問です．

る．プログラムのレベルが多層であることを区別しないと，かなりまずい．しかも人によって階層のとらえ方がばらばらであったり，「抽象化」が苦手な人が多いため，具体的なレベルへの落とし込みも重要になる．

パターンにも規模がある

開発する案件に応じて，いろいろな意味で規模があり，それぞれに対応した開発スタイルを取らないとやりにくくなります．個人レベルの「天才」プログラマが比較的多いのに対し，大規模レベルや企業レベルにおける天才プログラマの存在をほとんど聞かないのは，ほとんどの場合，個人レベルでの開発スタイルが詳しくわかっている（あるいは「わかっている」と本人が思い込んでいるだけ）のに対し，大規模レベルや企業レベルでは個人レベルでは想像もしなかったような事情が発生し，それに対応した開発スタイルが要求されるのに，そのスタイルがわかっていないからでしょう．

ところで最近は「デザインパターン」，あるいは単に「パターン」という考えがソフトウェア開発で現れていますが，これにも，それぞれの状況に応じた「規模」があり，その点があいまいだと「意思疎通」がうまく行きません．そもそもパターンとは何でしょうか？　昔からある考えで「アルゴリズム」というのがありますが，なかにはアルゴリズムとパターンを混同している人がいますし，両者が同一のものだと誤解している人もいます．本講座の最初に説明した「デメテルの法則」のような「法則」は，パターンやアルゴリズムとはどう違うのでしょう？　このあたりもあいまいにとらえている人がいるようなので，少し整理してみます．

定番

ある程度プログラミングを続けていると，以前にこなしたものと似通った手順で問題解決をはかっていることが多々あります．中には「あれ？これは以前にもあったぞ」という既視感（デジャブ）にとらわれることがあります．何度も遭遇する処理なら「ライブラリ」化するかもしれません．たとえば，C言語の文字列をコピーする処理は何度も遭遇するので，毎回，文字列のコピー処理をコーディングするのではなく標準ライブラリの「`strcpy`」を利用するのは誰でもやっていることでしょう．ところがライブラリ化できるものは具体性が高くて再利用しやすいように仕様をまとめたものだけで，具体性が低くあいまいな仕様のものをライブラリ化するのはきわめて難しいことです．そのため，すでにやったことがあると自覚しつつも似通ったコーディングをせっせと手間をかけて再度行うハメになります．いわゆる“車輪の再発明”と揶揄される状況です．

ここで「具体性」と語ったものは，仕様が明確であるかどうかというレベルだけでなく，特定のプログラミング言語やプログラミング環境への依存度をも含みます．あるプログラミング言語でしか通用しないテクニックは具体性が高いと考えられますが，汎用性は低いと考えられます．しかし即効性という点では，具体性が高いものほど，その状況に適した場面での有効度は高くなることでしょう．反面，適する状況がきわめて狭いため，それをカバーするためは「弾数（たまかず）」をたくさんもっていないと苦しくなる傾向があります．

いずれにせよ「以前にもあった」と判断し，それを解決する決まりきった手順（すなわち定番）があり，それらの手順には「具体性」に応じてレベル分けがあり，それぞれのレベルによって

定番の名称が変わっているわけです．たとえば，問題解決をするときに解く手順が機械的かつ具体的に実装しやすいものを「アルゴリズム」と呼んだり，特定のプログラミング言語における記述テクニックとして定義されているものを「イディオム」と呼ぶわけです．これは筆者独自の分類なのですが，さきほどの具体性に着目して3段階に分類すると，

① 具体性が高い：アルゴリズム，データ構造，イディオム，マクロ/関数ライブラリの使い方，形式知
② 具体性が中ぐらい：デザインパターン，オブジェクト指向設計，クラスライブラリ/フレームワークの使い方
③ 具体性が低い：アーキテクチャパターン（ソフトウェアを構築するさいに採用する構造組織化パターンのこと．ミクロな実装レベルではなく，プログラム全体のマクロな構造に着目したときに観察されるパターン），法則，定理，言い伝えや伝承の類い，暗黙知

となります．

アーキテクチャパターン

筆者流の分類で「アーキテクチャパターン」という耳慣れない

Column2

RAIIイディオム

RAIIイディオムの原理は，リストAのような実装をしたクラスを利用することです．このクラスで自動変数を記述すると，その場所で資源が確保され，変数の有効なスコープからはずれると資源が解放されます．こんな単純なものを使うメリットなどあるのかと疑う人がいるかもしれません．たとえばリストBを見てください．ここでは「char aMem[HUGE_SIZE];」という配列が自動変数で作成できないので，苦肉の策として動的に領域を確保しているのですが，f()の最後の行で解放を行っています．いうまでもなく，このコードでは途中リターンや例外によって解放されずにメモリリークする可能性があります．RAIIイディオムを使って，このコードを書き直すと，リストCのようになります．この場合，途中リターンや例外があっても必ずXMallocのデストラクタが呼び出されるので，メモリリークは起こりません．

〔リストB〕RAIIが必要な状況

```
#include <iostream>
#include <cstdlib>
#include <exception>
#include <stdexcept>
...(略)...
static bool checkNG();

static void g(char *iMem)
{
    ...(略)...
    if(checkNG()){
        throw std::runtime_error("check NG");
    }
    ...(略)...
}

static void f()
{
    char *aMem = static_cast<char *>(std::malloc(HUGE_SIZE));
    ...(略)...
    g(aMem); //問題あり！ 例外を出す場合がある.
    ...(略)...
    if(checkNG()){
        return; //問題あり！ 途中リターンしている.
    }
    ...(略)...
    std::free(aMem);
}
```

いうまでもありませんがRAIIイディオムは資源の確保/解放だけに限らず，ファイルのオープン/クローズや，排他制御のためのロック/アンロックにも応用できます．応用範囲はプログラマが想像つく限りはいろいろとありそうです．その意味で，C++を使う人にとってはRAIIイディオムは重要なイディオムといえるでしょう．

〔リストA〕RAIIイディオムの原理

```
class XClass {
public:
    XClass(){ //コンストラクタ
        ...(ここで資源を確保する)...
    }
    ~XClass(){ //デストラクタ
        ...(ここでコンストラクタで確保した資源を解放する)...
    }
};
```

〔リストC〕RAIIでメモリリークを防止

```
template <class T>
class XMalloc {

    void *mMem;

    XMalloc(); //(メモリサイズの指定を必ずさせるための工夫)

public:
    XMalloc(size_t iMemSize){
        mMem = std::malloc(iMemSize * sizeof(T));
    }

    ~XMalloc(){
        std::free(mMem);
    }

    operator T*(){
        return static_cast<T*>(mMem);
    }
};

static void f()
{
    XMalloc<char> aMem(HUGE_SIZE);
    ...(略)...
    g(aMem);
    ...(略)...
    if(checkNG()){
        return;
    }
    ...(略)...
}
```

ものが出ましたが，じつは *Pattern-Oriented Software Architecture*（http://www.posa.uci.edu/を参照．日本では『ソフトウェア アーキテクチャ』というタイトルで翻訳されている．こちらはhttp://www.kindaikagaku.co.jp/bookdata/ISBN4-7649-0283-4.htmを参照のこと），通称「POSA本」と呼ばれている書籍で解説されています．いわゆるデザインパターンが一つのソフトウェア製品内あるいは製品の局所で観察されるパターンであるのに対し，アーキテクチャパターンは製品全体や，あるいは複数製品を組み合わせて利用する場合に観察されるマクロレベルのパターンであるという違いがあります．POSA本に関しては本連載の先で取り上げるかもしれません．ただ本連載は，これより具体性の高いパターンや定番について論じていくので，しばらくは取り上げないことにします．

イディオム

イディオムというのは，特定のプログラミング言語に依存する記述でよく使う定番のことです．POSA本では「一つのプログラミング言語に特化した抽象度の低いパターンであり，コンポーネントとコンポーネント間の関係を，ある言語で実装する方法を表現する」と書いています．「抽象度が低い」というのが曲者（くせもの）で，一見デザインパターンと同じようなレベルと勘違いする危険があります．

たとえば，C++で「RAIIイディオム（RAIIとは“Resource Acquisition is Initialization”の頭文字が由来．『プログラミング言語C++第3版』の14.4「資源管理」で“資源確保は初期設定”と紹介されているテクニックのこと）」というイディオムがあります．これは自動変数としてインスタンスが記述された場合，そのインスタンスのコンストラクタが記述場所で呼び出され，スコープから抜け出した時点でデストラクタが自動的に呼び出される仕様を利用したテクニックで，具体的にはauto_ptrを実装するときに利用されているものです．しかし，この性質はC++だからこそできるテクニックであり，C++以外でも使えるという保証がありません（Visual Basic 6では使えるらしい）．

〔リスト1〕Javaでの Singletonを作るイディオム

```
class XClass {
    private static XClass singleton = null;

    public static XClass getSingleton(){
        if(singleton == null){
            singleton = new XClass();
        }
        return singleton;
    }

    private XClass(){
//必ずgetSingletonメソッドしか使えないようにする工夫
        ...(略)...
    }
    ...(略)...
}
```

その意味で「汎用性」に欠けており，抽象度が低いと評されるわけです．しかし，おもしろいといっては何ですが，POSA本では，デザインパターンとイディオムとの間に明確な境界線を引くのが難しいとも記述しています．というのも，あるデザインパターンを実装するための記述テクニックとして利用されるイディオムがあるからです．たとえばSingletonパターンというデザインパターンがありますが，Javaだとおそらく，リスト1のような記述が定番となるでしょう．しかし，よく考えると他のプログラミング言語でも，リスト2のような記述は定番パターンであり，そこだけをピックアップするなら，この記述パターンは抽象度が高いと評価できるかもしれません．

アルゴリズム

コンピュータを使って問題解決をするときに解く手順を，機械的な手順として準備する必要があります．この手順のことをアルゴリズムと称します．パターンとアルゴリズムの違いですが，POSA本によれば，

- パターンはアーキテクチャに照準を合わせている
- アルゴリズムはサーチやソートなどの個々の計算問題を扱う

と述べています（8.4「アルゴリズム，データ構造，パターン」を参照のこと）．パターンはソフトウェアがどういう構造を取るのかに関心を示すのに対し，アルゴリズムは個々の問題解決方法をどう取り扱うかに関心を示すということです．パターンはソフトウェア構築上の戦略や戦術レベルを意識するのに対し，アルゴリズムは局所における格闘技術や戦技を意識するわけです．

パターンやアーキテクチャはどちらかというと開発規模の影響を受けやすい傾向があります．個人レベルで最適なパターンやアーキテクチャと，企業レベルでのそれが食い違うことがあるかもしれません．しかしながら，個人レベルで扱うアルゴリズムと企業レベルで扱うアルゴリズムにさほど差があるとは思えません．どういう開発規模であっても，アルゴリズム自体が影響を受けるわけではありません．個人が趣味レベルでささやかに行うプログラムで採用するアルゴリズムと，大規模プロジェクトで採用するアルゴリズムには，ほとんど変わりはないのです．こんなことをいうと，個人が扱うデータ量と企業が扱うデータ量には差があるからアルゴリズムにも差があるだろうという反論があるのですが，注意してほしいのは，要求される

〔リスト2〕Singletonを作る記述パターン

```
外部非公開変数 singleton = 無効値

singletonの型 singletonを取得するメソッド {
    if(singletonが無効値である){
        singleton = singletonを発生
    }
    singletonを返す
}
```

案件によってデータ量が変わってくるということです．企業レベルの開発でも超小型の組み込み機器で扱う量は知れていますし，個人の開発でもGバイト単位のシミュレーションプログラムを作るとなると，それに適したアルゴリズムは巨大な規模を要求されるからです．そういう意味でアルゴリズムを勉強することはどの開発レベルであっても重要ですし，パターンの勉強とは別にやっておく値打ちがあります．このごろはパターンがトレンドだからアルゴリズムなんて無視してしまえ，とは決してならないのです．

Mastering Algorithms with C

本連載ではしばらくアルゴリズムの話をする予定ですが，困ったことに（?!）アルゴリズムに関する書籍は多数あり，そのどれもが一長一短があり，なかなかこれと思うものが見当たらないというのが実状です．アルゴリズムはイディオムとは違い，あまり特定のプログラミング言語の影響を受けないはずなのですが，現状出ているものはたいてい手続き指向言語を前提にしているものばかりです（最近出た，あるJavaを使ったアルゴリズム事典を読んでいて驚いたのは，メソッドに`public static`を付けて手続き指向風にしていたことである．おそらくはすでに出ていた手続き言語版をJavaに焼き直しただけなのだろうが，ちょっと安直なのではないかと思った）．そのため，そのまま利用すると手続き指向の延長線上で問題解決をはかる危険があるため，なるべくそのような危険を除去できるものをと考えるのですが，なかなか見当たりません．ところが1999年にO'Reilly社から出版された*Mastering Algorithms with C*（`http://www.oreilly.com/catalog/masteralgoc/`を参照）というのが，なかなかよく書けていたのです．「C言語を使って」とありますが，オブジェクト指向言語に移植することを配慮しているフシがありますし，取り上げている話題の範囲もまんべんなく，しかもさほどマニアックな方向に向かっていないので（ようするに，現実のソフトウェア開発でめったに使いそうもない難解かつ特殊なアルゴリズムをとりあげていないということ），これを教材として，しばらくおつきあいいただきたいと思います．

Column3

JavaでSingletonを作るイディオム

本文で説明したSingletonを作るイディオムだと，マルチスレッド環境では破綻します．というのも，

```
    if(singleton == null){ //(1)
        singleton = new XClass(); //(2)
```

というコードがあったとき，(1)を通過したスレッドが(2)に取りかかろうとした直後はまだ`singleton`は`null`のままです．そのため運悪く，たまたま同時に(1)を通過する別スレッドがあった場合，そのスレッドも(2)に突入します．結果的に2回「`new XClass()`」が実行され，Singletonが実現できなくなります．これを防止するため，リストDのような記述をするのが定番であり，これを「double-checked lockingイディオム」と称します．このイディオムを使うと，先ほどの二重通過が防止されます．というのも，あるスレッドが(2)を通過したときに運悪く別のスレッドも突入してきたとしても，(2)の`synchronized`のおかげで同時に二つのスレッドが(3)，(4)を実行することはなく，片方は(2)で待たされることになります．最初に通過したスレッドが(5)から先にいくと，待たされていたスレッドは(3)にいきますが，このとき`singleton`は`null`ではないので(4)を通過しません．したがって，2回「`new XClass()`」されることはありません．ところが困ったことに，現状のJavaの実装ではうまくいかないことが「double-checked lockingとSingletonパターン」（注：`http://www-6.ibm.com/jp/developerworks/java/020726/j_j-dcl.html`を参照）という記事で説明されています．

getSingletonメソッドをsynchronizedメソッドにしてしまうのがもっとも安直ですが，そもそもdouble-checked lockingイディオムはsynchronizedによる実行効率低下を防止するテクニックとして考案されたものなので，実行効率低下を気にするならスタティックイニシャライザを使うのが手っとり早いでしょう．たとえば，リストEのように記述します．

〔リストD〕double-checked lockingイディオム

```
class XClass {
    private static XClass singleton = null;

    public static XClass getSingleton(){
        if(singleton == null){ //(1)
            synchronized(XClass.class){ //(2)
                if(singleton == null){  //(3)
                    singleton = new XClass(); //(4)
                }
            } //(5)
        }
        return singleton;
    }
    ...(略)...
}
```

〔リストE〕スタティックイニシャライザを利用したSingleton

```
class XClass {

    private static XClass singleton;

    static {
        singleton = new XClass();
    }

    public static XClass getSingleton(){
        return singleton;
    }
    ...(略)...
}
```

Mastering Algorithms with C の目次によれば，次のような内容が取り上げられています．

- Recursion ── 再帰
- Linked Lists ── 連結リスト
- Stack ──スタック
- Queue ──キュー
- Sets ── 集合
- Hash Tables ── ハッシュテーブル
- Trees ── 木構造
- Heap ── ヒープ（アルゴリズムの本でヒープといった場合，自由メモリ領域のことではなく，データを高速に整列済みにする構造のこと．連載の先のほうで詳しく説明する予定）
- Priority Queue ── 優先順位を意識したキュー
- Graphs ── グラフ

ここまでは第1部と第2部で，どちらかというとデータ構造にかかわる内容です．第3部は，次のような内容が取り上げられています．

- Sorting and Searching ── ソートと検索
- Numerical Methods ── 計算式
- Data Compression ── データ圧縮
- Data Encryption ── 暗号化
- Graph Algorithms ── グラフを使ったアルゴリズム（最短距離を求める手法など）
- Geometric Algorithms ── 図形に関するアルゴリズム

難しそうな内容のものもありますが，感心するのは，きちんと実装コードを示している点です．日本語訳がないのはじつに残念といわざるを得ません．アルゴリズムの本でありがちなのは，難しい話を難しく書き，そのくせ，どのように実装すべきかが明確に書かれていないため，読んだはいいが後に何も残らなかったり，どうやって応用させていいのか頭をかかえるものもいくつかあります．

それから本連載ではコードをそのまま引用するといろいろと差し障りがあると判断し，また現状は手続き指向に偏って考えてしまいがちなアルゴリズムの考えをもっと自由にとらえてもらえるよう，あえてJavaでコードを書き直してみます．C言語レベルでの実装に関しては「Mastering Algorithms with C」を参照していただくか，C言語で実装されているアルゴリズムの書籍を参考にしていただきたいと思います．

みやさか・でんと　miyadent@anet.ne.jp

DCT，IDCTの効率的構成法



三谷 政昭

前回（2003年12月号）は「DCTとマルチレート信号処理」と題し，信号のサンプリング周波数を可変（間引き，補間）することによって，DCT，IDCT処理を実現する手法について解説した．今回は，サブバンド，マルチレート信号処理をさらに進め，DCT，IDCT処理の効率的構成法として，フィルタバンクの“ツリー構成”と呼ばれるシステムを紹介する．本システムは，基本となるローパスフィルタとハイパスフィルタを直並列的に接続することにより，信号成分を複数の周波数帯域に分割してDCT，IDCT処理を実現するものである．なお，ツリー構成の考え方はウェーブレット変換に直接つながるものなので，しっかりと理解してもらいたい．

（筆者）

DCT，IDCTにおけるフィルタバンクの効率的構成法

前回は，DCT/IDCT計算を実現する手法として，サブバンドとマルチレート信号処理に基づく「フィルタバンク構成システム」を紹介した（図20.1）．本信号処理システムは，①分析（アナリシスバンク；Analysis bank），②合成（シンセシスバンク；Synthesis bank）の二つのフィルタバンクから構成され，分析フィルタバンクの出力はサブバンド信号（DCT値に相当）と呼ばれ，周波数帯域ごとに信号が分割されている（具体例として二つの帯域分割による構成を示した）．

それでは，フィルタバンクにおいて，サブバンド信号として3分割以上の帯域分割を行う場合のシステム構成の考え方を説明する．まずは，4サンプルを一つのブロックとしてDCT/IDCT処理する場合を例にとり，信号成分を3分割以上の周波数帯域に分割し，得られた信号から元の信号を再合成する処理を体験してみる（図20.2，図20.3）．これらのフィルタバンクのシステム構成は，ちょうど樹木（英語でtree，以下ではツリーと表記）の枝分かれしているようすに似ており，図20.2は“等分割によるツリー構成”，図20.3は“オクターブ分割によるツリー構成”と呼ばれる．

等分割によるツリー構成

それでは図20.2のシステム構成（周波数成分を4分割）について，分析と合成の各フィルタバンクにおける信号計算の流れを追ってみよう（図20.4）．

● 分析フィルタバンク

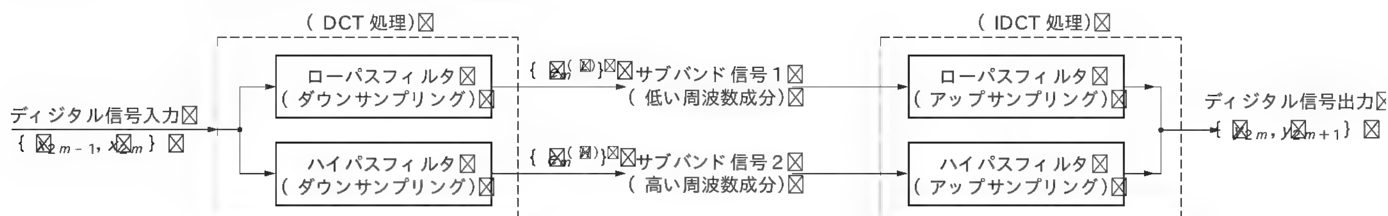
いまディジタル信号を，

$$\{x_{-3},\ x_{-2},\ x_{-1},\ x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ \cdots\}$$

と左から右へと順に4サンプルを1ブロックとして順に入力して得られる二つの周波数帯域に分割された信号をそれぞれ，

$$\{e_{-1}^{(L)},\ e_0^{(L)},\ e_1^{(L)},\ e_2^{(L)},\ e_3^{(L)},\ e_4^{(L)},\ \cdots\} \quad \cdots\cdots(1)$$

〔図20.1〕フィルタバンクの2分割によるシステム構成例





$$\{ e_{-1}^{(H)},\ e_0^{(H)},\ e_1^{(H)},\ e_2^{(H)},\ e_3^{(H)},\ e_4^{(H)},\ \cdots \} \quad \cdots(2)$$

としよう(第1段目)．ここで，式(1)は上付き文字(L)で第1段目のローパスフィルタの出力を，式(2)は上付き文字(H)で第1段目のハイパスフィルタの出力を表している．以降では，ローパスフィルタの出力を“LP出力”，ハイパスフィルタの出力を“HP出力”と表記することにする．さらに，式(1)のLP出力$\{e_m^{(L)}\}$から得られる二つの周波数帯域に分割された信号をそれぞれ，

$$\{ e_0^{(LL)},\ e_1^{(LL)},\ e_2^{(LL)},\ e_3^{(LL)},\ e_4^{(LL)},\ e_5^{(LL)},\ \cdots \} \quad \cdots(3)$$

$$\{ e_0^{(LH)},\ e_1^{(LH)},\ e_2^{(LH)},\ e_3^{(LH)},\ e_4^{(LH)},\ e_5^{(LH)},\ \cdots \} \quad \cdots(4)$$

とし，式(2)のHP出力$\{e_m^{(H)}\}$から得られる二つの周波数帯域に分割された信号をそれぞれ，

$$\{ e_0^{(HL)},\ e_1^{(HL)},\ e_2^{(HL)},\ e_3^{(HL)},\ e_4^{(HL)},\ e_5^{(HL)},\ \cdots \} \quad \cdots(5)$$

$$\{ e_0^{(HH)},\ e_1^{(HH)},\ e_2^{(HH)},\ e_3^{(HH)},\ e_4^{(HH)},\ e_5^{(HH)},\ \cdots \} \quad \cdots(6)$$

と表すことにする(第2段目)．ここで式(3)と式(5)は第2段目のLP出力，式(4)と式(6)は第2段目のHP出力に相当している(図20.4)．このとき，DCT値は次のようにブロック計算される．

① $\{ x_{-3},\ x_{-2},\ x_{-1},\ x_0 \}$に対する計算

● 第1段目

$$e_{-1}^{(L)} = \frac{x_{-2}+x_{-3}}{2},\quad e_0^{(L)} = \frac{x_0+x_{-1}}{2} \quad \cdots\cdots(7)$$

$$e_{-1}^{(H)} = \frac{-x_{-2}+x_{-3}}{2},\quad e_0^{(H)} = \frac{-x_0+x_{-1}}{2} \quad \cdots\cdots(8)$$

● 第2段目

$$e_0^{(LL)} = \frac{e_0^{(L)}+e_{-1}^{(L)}}{2} \quad \cdots\cdots(9)$$

$$e_0^{(LH)} = \frac{-e_0^{(L)}+e_{-1}^{(L)}}{2} \quad \cdots\cdots(10)$$

$$e_0^{(HL)} = \frac{e_0^{(H)}+e_{-1}^{(H)}}{2} \quad \cdots\cdots(11)$$

$$e_0^{(HH)} = \frac{-e_0^{(H)}+e_{-1}^{(H)}}{2} \quad \cdots\cdots(12)$$

② $\{ x_1,\ x_2,\ x_3,\ x_4 \}$に対する計算

● 第1段目

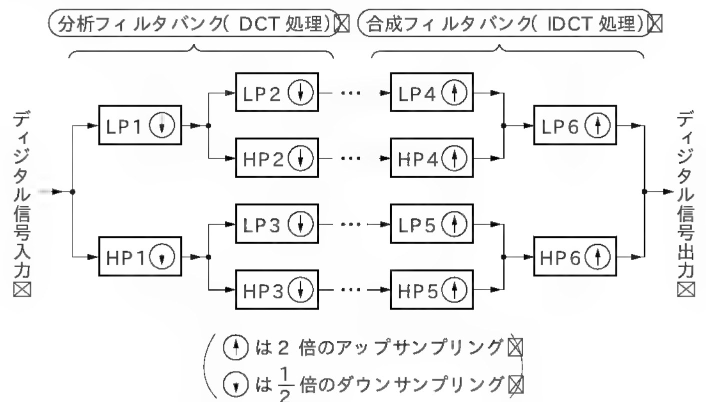
$$e_1^{(L)} = \frac{x_2+x_1}{2},\quad e_2^{(L)} = \frac{x_4+x_3}{2} \quad \cdots\cdots(13)$$

$$e_1^{(H)} = \frac{-x_2+x_1}{2},\quad e_2^{(H)} = \frac{-x_4+x_3}{2} \quad \cdots\cdots(14)$$

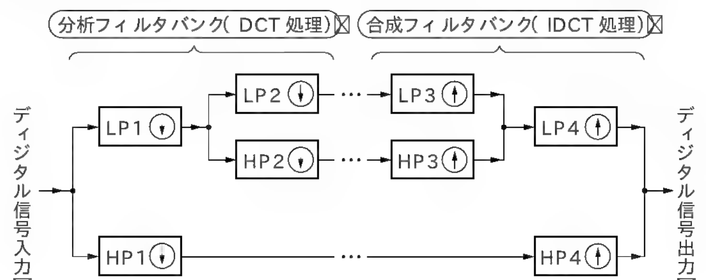
● 第2段目

$$e_1^{(LL)} = \frac{e_2^{(L)}+e_1^{(L)}}{2} \quad \cdots\cdots(15)$$

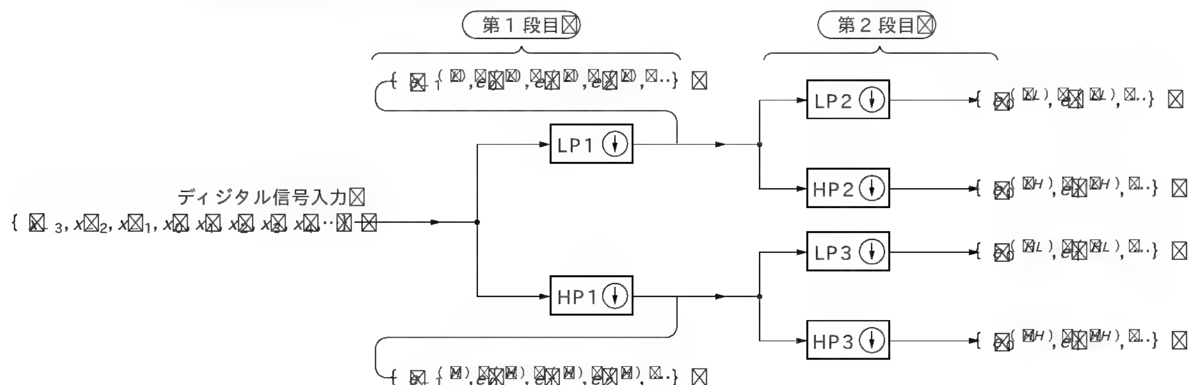
〔図20.2〕2分割フィルタバンクのツリー構成による等分割の例



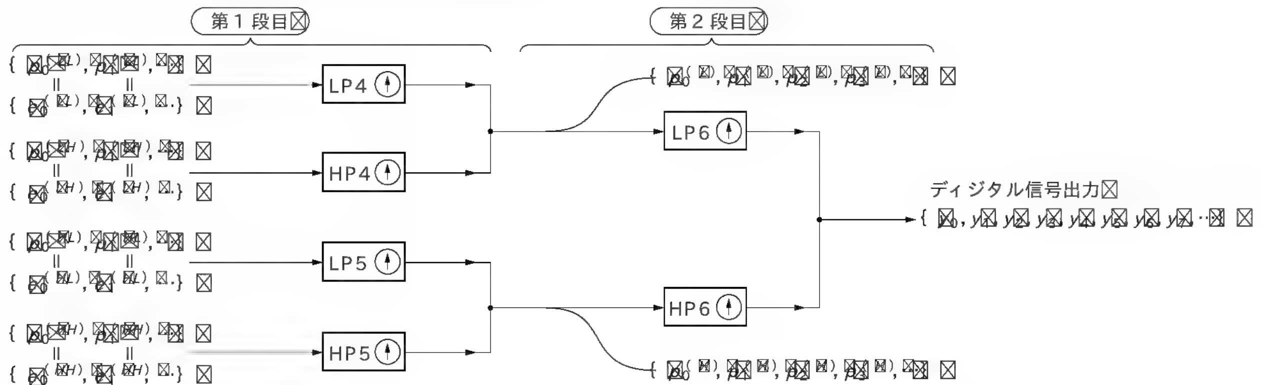
〔図20.3〕2分割フィルタバンクのツリー構成によるオクターブ分割の例



〔図20.4〕分析フィルタバンクの信号計算の流れ(等分割の場合)



〔図20.5〕合成フィルタバンクの信号計算の流れ(等分割の場合)



$$e_{1\boxtimes}^{(LH)} = \frac{-e_{2\boxtimes}^{(L)} + e_{1\boxtimes}^{(L)}}{2\boxtimes} \quad \cdots\cdots (16)$$

$$e_{1\boxtimes}^{(HL)} = \frac{e_{2\boxtimes}^{(H)} + e_{1\boxtimes}^{(H)}}{2\boxtimes} \quad \cdots\cdots (17)$$

$$e_{1\boxtimes}^{(HH)} = \frac{-e_{2\boxtimes}^{(H)} + e_{1\boxtimes}^{(H)}}{2\boxtimes} \quad \cdots\cdots (18)$$

● 合成フィルタバンク

次に，合成フィルタバンク(IDCT値の計算に相当)のシステム構成を図20.5に示す．ここで，説明の便宜上，分析フィルタバンクの四つの周波数帯域に分割された出力の変数名 e を p と書き換えることにし，式(3)～式(6)をそれぞれ，

$$\{p_0^{(LL)},\ p_1^{(LL)},\ p_2^{(LL)},\ p_3^{(LL)},\ p_4^{(LL)},\ p_5^{(LL)},\ \cdots\} \quad \cdots\cdots (19)$$

$$\{p_0^{(LH)},\ p_1^{(LH)},\ p_2^{(LH)},\ p_3^{(LH)},\ p_4^{(LH)},\ p_5^{(LH)},\ \cdots\} \quad \cdots\cdots (20)$$

$$\{p_0^{(HL)},\ p_1^{(HL)},\ p_2^{(HL)},\ p_3^{(HL)},\ p_4^{(HL)},\ p_5^{(HL)},\ \cdots\} \quad \cdots\cdots (21)$$

$$\{p_0^{(HH)},\ p_1^{(HH)},\ p_2^{(HH)},\ p_3^{(HH)},\ p_4^{(HH)},\ p_5^{(HH)},\ \cdots\} \quad \cdots\cdots (22)$$

としよう．式(19)～式(22)は合成フィルタバンクの第1段目の入力であり，ローパスフィルタの出力を，

$$\{p_0^{(L)},\ p_1^{(L)},\ p_2^{(L)},\ p_3^{(L)},\ p_4^{(L)},\ p_5^{(H)},\ \cdots\} \quad \cdots\cdots (23)$$

とし，ハイパスフィルタの出力を，

$$\{p_0^{(H)},\ p_1^{(H)},\ p_2^{(H)},\ p_3^{(H)},\ p_4^{(H)},\ p_5^{(H)},\ \cdots\} \quad \cdots\cdots (24)$$

と表すことにする．さらに，第2段目の出力を，

$$\{y_0,\ y_1,\ y_2,\ y_3,\ y_4,\ y_5,\ y_6,\ y_7,\ \cdots\} \quad \cdots\cdots (25)$$

と表せば，IDCT値の出力として図20.5のようにブロック計算される．

① $\{p_0^{(LL)},\ p_0^{(LH)},\ p_0^{(HL)},\ p_0^{(HH)}\}$ に対する計算

● 第1段目

$$p_0^{(L)} = p_0^{(LL)} + p_0^{(HL)},\ p_1^{(L)} = p_0^{(LL)} - p_0^{(LH)} \quad \cdots\cdots (26)$$

$$p_0^{(H)} = p_0^{(HL)} + p_0^{(HH)},\ p_1^{(H)} = p_0^{(HL)} - p_0^{(HH)} \quad \cdots\cdots (27)$$

● 第2段目

$$\begin{cases} y_0 = p_0^{(L)} + p_0^{(H)} \\ y_1 = p_0^{(L)} - p_0^{(H)} \\ y_2 = p_1^{(L)} + p_1^{(H)} \\ y_3 = p_1^{(L)} - p_1^{(H)} \end{cases} \quad \cdots\cdots (28)$$

② $\{p_1^{(LL)},\ p_1^{(LH)},\ p_1^{(HL)},\ p_1^{(HH)}\}$ に対する計算

● 第1段目

$$p_2^{(L)} = p_1^{(LL)} + p_1^{(LH)},\ p_3^{(L)} = p_1^{(LL)} - p_1^{(LH)} \quad \cdots (29)$$

$$p_2^{(H)} = p_1^{(HL)} + p_1^{(HH)},\ p_3^{(H)} = p_1^{(HL)} - p_1^{(HH)} \quad \cdots (30)$$

● 第2段目

$$\begin{cases} y_{4\boxtimes} = p_{2\boxtimes}^{(L)} + p_{2\boxtimes}^{(H)} \\ y_{5\boxtimes} = p_{2\boxtimes}^{(L)} - p_{2\boxtimes}^{(H)} \\ y_{6\boxtimes} = p_{3\boxtimes}^{(L)} + p_{3\boxtimes}^{(H)} \\ y_{7\boxtimes} = p_{3\boxtimes}^{(L)} - p_{3\boxtimes}^{(H)} \end{cases} \quad \cdots\cdots (31)$$

例題1

式(7)～式(31)で処理されるDCT，IDCT計算において，図20.6に示すように，m 番目のブロックの入力信号を，

$$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$$

とし，合成フィルタバンクの m 番目のブロックの出力信号を，

$$\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$$

とするとき，分析と合成の各フィルタバンクでの信号処理の一般式を示せ(図20.6)．

解答1

① 分析フィルタバンク

● 第1段目

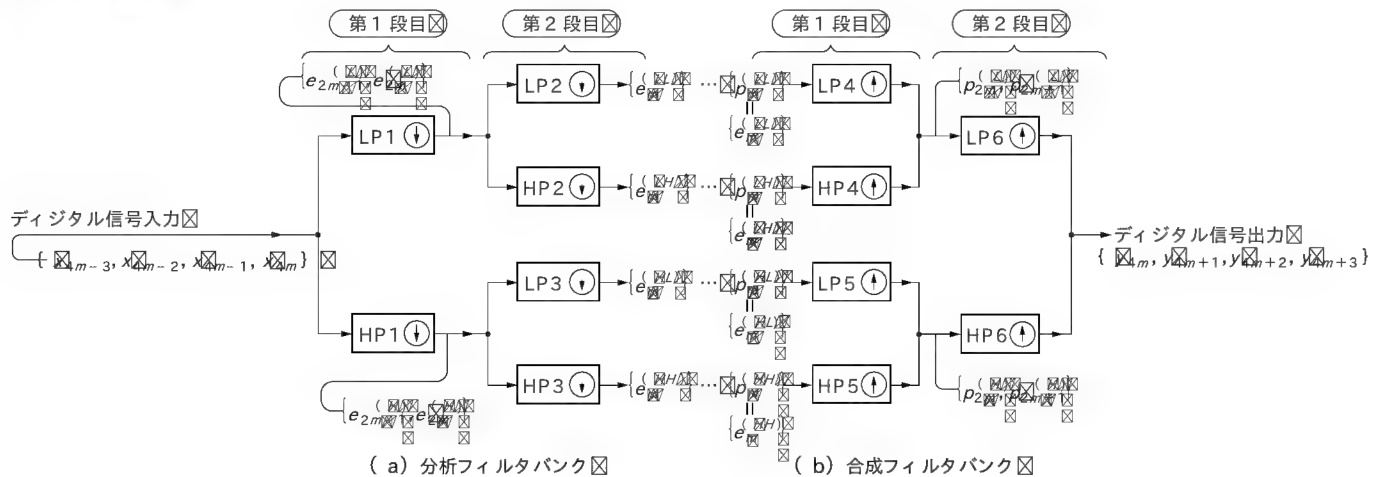
式(7)，式(8)，式(13)，式(14)を参照．

$$e_{2\boxtimes-1\boxtimes}^{(L)} = \frac{x_{4\boxtimes-2\boxtimes} + x_{4\boxtimes-3\boxtimes}}{2\boxtimes},\ e_{2\boxtimes}^{(L)} = \frac{x_{4\boxtimes} + x_{4\boxtimes-1\boxtimes}}{2\boxtimes} \quad \cdots (32)$$

$$e_{2\boxtimes-1\boxtimes}^{(H)} = \frac{-x_{4\boxtimes-2\boxtimes} + x_{4\boxtimes-3\boxtimes}}{2\boxtimes},\ e_{2\boxtimes}^{(H)} = \frac{-x_{4\boxtimes} + x_{4\boxtimes-1\boxtimes}}{2\boxtimes} \quad \cdots (33)$$



〔図20.6〕 例題1



（a）分析フィルタバンク　（b）合成フィルタバンク

● 第2段目

式（9）～式（12），式（15）～式（18）を参照．

$$e_m^{(LL)} = \frac{e_{2m}^{(L)} + e_{2m-1}^{(L)}}{2} \quad \cdots\cdots (34)$$

$$e_m^{(LH)} = \frac{-e_{2m}^{(L)} + e_{2m-1}^{(L)}}{2} \quad \cdots\cdots (35)$$

$$e_m^{(HL)} = \frac{e_{2m}^{(H)} + e_{2m-1}^{(H)}}{2} \quad \cdots\cdots (36)$$

$$e_m^{(HH)} = \frac{-e_{2m}^{(H)} + e_{2m-1}^{(H)}}{2} \quad \cdots\cdots (37)$$

式（32）～式（37）の一般式に示すように，入力信号と出力信号をそれぞれ4サンプルごとに一つのブロックにまとめて計算する手法が，DCT/IDCT 計算の効率的な処理の実現を可能にするのであり，ツリー構成で表される．ここで，式（32）～式（37）の計算処理を簡単に説明しておく．

［式（32），式（33）］：「4サンプルの信号 $\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$ が第1段目のLP出力 $\{e_m^{(L)}\}$，あるいはHP出力 $\{e_m^{(H)}\}$ として，2サンプルにダウンサンプリングされているようす」

［式（34），式（35）］：「2サンプルの第1段目のLP出力 $\{e_m^{(L)}\}$ が，さらに第2段目のLP出力 $\{e_m^{(LL)}\}$，あるいはHP出力 $\{e_m^{(LH)}\}$ として，1サンプルにダウンサンプリングされているようす」

［式（36），式（37）］：「2サンプルの第1段目のHP出力 $\{e_m^{(H)}\}$ がさらに第2段目のLP出力 $\{e_m^{(HL)}\}$，あるいはHP出力 $\{e_m^{(HH)}\}$ として，1サンプルにダウンサンプリングされているようす」

なお，式（32）～式（37）で表す分析フィルタバンクの一般式において，$m=0$ とすれば式（7）～式（12），$m=1$ とすると式（13）～式（18）が導かれることを各自で検証してもらいたい．

② 分析フィルタバンク

説明の便宜上，分析フィルタバンクの出力信号 $\{e_m^{(LL)}\}$，$\{e_m^{(LH)}\}$，$\{e_m^{(HL)}\}$，$\{e_m^{(HH)}\}$ を，次のように表すことにする．

$$\begin{cases} p_m^{(LL)} = e_m^{(LL)},\ p_m^{(LH)} = e_m^{(LH)} \\ p_m^{(HL)} = e_m^{(HL)},\ p_m^{(HH)} = e_m^{(HH)} \end{cases} \quad \cdots\cdots (38)$$

● 第1段目

式（26），式（27），式（29），式（30）を参照．

$$\{p_{2m}^{(L)} = p_m^{(LL)} + p_m^{(LH)},\ p_{2m+1}^{(L)} = p_m^{(LL)} - p_m^{(LH)}\} \quad \cdots\cdots (39)$$

$$\{p_{2m}^{(H)} = p_m^{(HL)} + p_m^{(HH)},\ p_{2m+1}^{(H)} = p_m^{(HL)} - p_m^{(HH)}\} \quad \cdots\cdots (40)$$

● 第2段目

式（28），式（31）を参照．

$$\begin{cases} y_{4m} = p_{2m}^{(L)} + p_{2m}^{(H)} \\ y_{4m+1} = p_{2m}^{(L)} - p_{2m}^{(H)} \\ y_{4m+2} = p_{2m+1}^{(L)} + p_{2m+1}^{(H)} \\ y_{4m+3} = p_{2m+1}^{(L)} - p_{2m+1}^{(H)} \end{cases} \quad \cdots\cdots (41)$$

以上より，式（39）～式（41）は以下のような処理を表していることが理解される．

［式（39）］：「1サンプルの信号 $\{p_m^{(LL)}\}$ と $\{p_m^{(LH)}\}$ から，第1段目のLP出力，あるいはHP出力として，2サンプルの信号 $\{p_m^{(L)}\}$ にアップサンプリングされているようす」

［式（40）］：「1サンプルの信号 $\{p_m^{(HL)}\}$ と $\{p_m^{(HH)}\}$ から，第1段目のLP出力，あるいはHP出力として，2サンプルの信号 $\{p_m^{(H)}\}$ にアップサンプリングされているようす」

［式（41）］：「2サンプルの第1段目のLP出力 $\{p_m^{(L)}\}$ とHP出力 $\{p_m^{(H)}\}$ から，第2段目のLP出力，あるいはHP出力として，4サンプルの信号 $\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$ にアップサンプリングされているようす」

なお，式（39）～式（41）で表す合成フィルタバンクの一般式において，$m=0$ とすれば式（26）～式（28），$m=1$ とすると式（29）～式（31）が導かれることを各自で検証してもらいたい．

●“完全再構成条件”

まず，式(32)，式(33)を式(34)～式(37)に代入することにより，“分析フィルタバンク”の四つの周波数帯域に分割された出力は，それぞれ4サンプルの信号$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$を用いて次のように表される．

$$e_m{}^{(LL)} = \frac{x_{4m} + x_{4m-1} + x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots(42)$$

$$e_m{}^{(LH)} = \frac{-x_{4m} - x_{4m-1} + x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots(43)$$

$$e_m{}^{(HL)} = \frac{-x_{4m} + x_{4m-1} - x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots(44)$$

$$e_m{}^{(HH)} = \frac{x_{4m} - x_{4m-1} - x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots(45)$$

さらに続けて，式(42)～式(45)を式(38)～式(41)に代入することにより，“合成フィルタバンク”の4サンプルの出力信号$\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$が次のように導かれる．

●第1段目

$$p_{2m}{}^{(L)} = \frac{x_{4m-2} + x_{4m-3}}{2},\quad p_{2m+1}{}^{(L)} = \frac{x_{4m} + x_{4m-1}}{2} \quad \cdots\cdots(46)$$

$$p_{2m}{}^{(H)} = \frac{-x_{4m-2} + x_{4m-3}}{2},\quad p_{2m+1}{}^{(H)} = \frac{-x_{4m} + x_{4m-1}}{2} \quad \cdots\cdots(47)$$

●第2段目

$$\begin{cases} y_{4m} = p_{2m}{}^{(L)} + p_{2m}{}^{(H)} = \dfrac{2x_{4m-3}}{2} = x_{4m-3} \\ y_{4m+1} = p_{2m}{}^{(L)} - p_{2m}{}^{(H)} = \dfrac{2x_{4m-2}}{2} = x_{4m-2} \\ y_{4m+2} = p_{2m+1}{}^{(L)} + p_{2m+1}{}^{(H)} = \dfrac{2x_{4m-1}}{2} = x_{4m-1} \\ y_{4m+3} = p_{2m+1}{}^{(L)} - p_{2m+1}{}^{(H)} = \dfrac{2x_{4m}}{2} = x_{4m} \end{cases} \quad \cdots\cdots(48)$$

以上より，式(42)～式(45)は“分析フィルタバンク”における周波数成分ごとに分割する信号処理，式(46)～式(48)は“合成フィルタバンク”における再合成する処理に相当することがわかる．よって，式(48)より，入力信号のブロック$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$と出力信号のブロック$\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$との間には，

$$\begin{cases} y_{4m} = x_{4m-3} \\ y_{4m+1} = x_{(4m+1)-3} = x_{4m-2} \\ y_{4m+2} = x_{(4m+2)-3} = x_{4m-1} \\ y_{4m+3} = x_{(4m+3)-3} = x_{4m} \end{cases} \quad \cdots\cdots(49)$$

となる関係が成立し，$k = 4m,\ 4m+1,\ 4m+2,\ 4m+3$として，

$$y_k = x_{k-3} \quad \cdots\cdots(50)$$

とまとめて表現でき，“完全再構成条件(入力信号が単にずれてそのまま出力される)”を満たしていることが確認される．つまり，入力信号系列が3サンプル時間だけ遅れて出力されることになり，等分割によるツリー構成によるフィルタバンクシステムの妥当性が理解される．

例題2

等分割によるツリー構成において，分析と合成フィルタバンクで用いるローパスフィルタとハイパスフィルタの係数の絶対値が同じになるDCT，IDCTの計算式を示せ．

解答2

フィルタバンク構成の実現条件に基づき，係数の絶対値が同じ，すなわち，$1/\sqrt{2}$となるものを選ぶ(2003年12月号「DCTとマルチレート信号処理」のpp.196-197参照)．次に計算式を示しておくので，“完全再構成条件(式(50))”を満たすことも確認してもらいたい．

●分析フィルタバンク

●第1段目

$$e_{2m-1}{}^{(L)} = \frac{x_{4m-2} + x_{4m-3}}{\sqrt{2}},\quad e_{2m}{}^{(L)} = \frac{x_{4m} + x_{4m-1}}{\sqrt{2}}$$

$$e_{2m-1}{}^{(H)} = \frac{-x_{4m-2} + x_{4m-3}}{\sqrt{2}},\quad e_{2m}{}^{(H)} = \frac{-x_{4m} + x_{4m-1}}{\sqrt{2}}$$

●第2段目

$$e_m{}^{(LL)} = \frac{e_{2m}{}^{(L)} + e_{2m-1}{}^{(L)}}{\sqrt{2}}$$

$$e_m{}^{(LH)} = \frac{-e_{2m}{}^{(L)} + e_{2m-1}{}^{(L)}}{\sqrt{2}}$$

$$e_m{}^{(HL)} = \frac{e_{2m}{}^{(H)} + e_{2m-1}{}^{(H)}}{\sqrt{2}}$$

$$e_m{}^{(HH)} = \frac{-e_{2m}{}^{(H)} + e_{2m-1}{}^{(H)}}{\sqrt{2}}$$

●合成フィルタバンク

分析フィルタバンクの出力を次のように表して，合成フィルタバンクの計算処理を記述する．

$$p_m{}^{(LL)} = e_m{}^{(LL)},\quad p_m{}^{(LH)} = e_m{}^{(LH)}$$
$$p_m{}^{(HL)} = e_m{}^{(HL)},\quad p_m{}^{(HH)} = e_m{}^{(HH)}$$

●第1段目

$$p_{2m}{}^{(L)} = \frac{p_m{}^{(LL)} + p_m{}^{(LH)}}{\sqrt{2}},\quad p_{2m+1}{}^{(L)} = \frac{p_m{}^{(LL)} - p_m{}^{(LH)}}{\sqrt{2}}$$

$$p_{2m}{}^{(H)} = \frac{p_m{}^{(HL)} + p_m{}^{(HH)}}{\sqrt{2}},\quad p_{2m+1}{}^{(H)} = \frac{p_m{}^{(HL)} - p_m{}^{(HH)}}{\sqrt{2}}$$

●第2段目

$$y_{4m} = y_0{}^{(m)} = \frac{p_{2m}{}^{(L)} + p_{2m}{}^{(H)}}{\sqrt{2}}$$

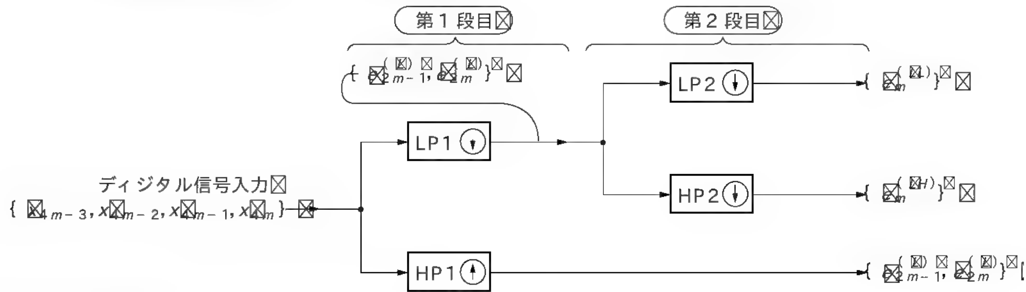
$$y_{4m+1} = y_1{}^{(m)} = \frac{p_{2m}{}^{(L)} - p_{2m}{}^{(H)}}{\sqrt{2}}$$

$$y_{4m+2} = y_2{}^{(m)} = \frac{p_{2m+1}{}^{(L)} + p_{2m+1}{}^{(H)}}{\sqrt{2}}$$

$$y_{4m+3} = y_3{}^{(m)} = \frac{p_{2m+1}{}^{(L)} - p_{2m+1}{}^{(H)}}{\sqrt{2}}$$



〔図20.7〕分析フィルタバンクの信号計算の流れ（オクターブ分割の場合）



オクターブ分割によるツリー構成

次に，図20.3の形式のシステム構成（周波数成分を3分割）について，分析と合成の各フィルタバンクにおける信号計算の流れを追ってみよう．図20.2の"等分割によるツリー構成"との違いは，分析フィルタバンクの第1段目のHP出力をさらに周波数分割するかどうかの一点であることに注意しつつ，読み進めていってほしい（図20.7）．

● 分析フィルタバンク

● 第1段目

まず，m番目のブロックのディジタル信号$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$から得られる二つの周波数帯域に分割された信号，すなわちLPの出力$\{e_m^{(L)}\}$とHPの出力$\{e_m^{(H)}\}$を次式で計算する．

$$e_{2m-1}^{(L)} = \frac{x_{4m-2}+x_{4m-3}}{2},\quad e_{2m}^{(L)} = \frac{x_{4m}+x_{4m-1}}{2} \quad \cdots (51)$$

$$e_{2m-1}^{(H)} = \frac{-x_{4m-2}+x_{4m-3}}{2},\quad e_{2m}^{(H)} = \frac{-x_{4m}+x_{4m-1}}{2} \quad \cdots (52)$$

● 第2段目

さらに続けて，第1段目のLP出力$\{e_m^{(L)}\}$だけを，

$$e_m^{(LL)} = \frac{e_{2m}^{(L)}+e_{2m-1}^{(L)}}{2} \quad \cdots\cdots (53)$$

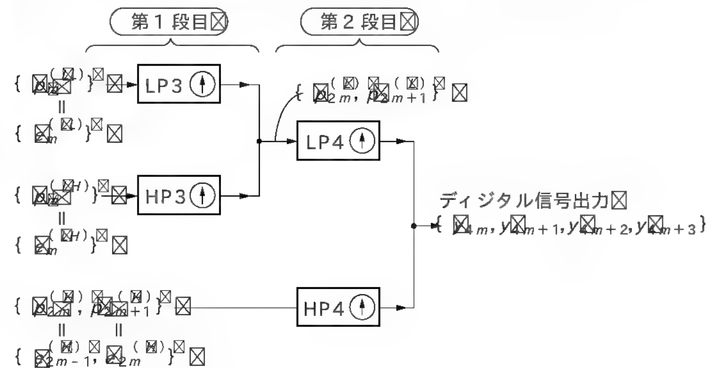
$$e_m^{(LH)} = \frac{-e_{2m}^{(L)}+e_{2m-1}^{(L)}}{2} \quad \cdots\cdots (54)$$

のようにLP出力$\{e_m^{(LL)}\}$，あるいはHP出力$\{e_m^{(LH)}\}$として二つに周波数分割し，DCT値を計算する．ここで，式（51）〜式（54）の計算処理を簡単に説明しておく．

［式（51），式（52）］:「4サンプルの信号$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$が第1段目のLP出力$\{e_m^{(L)}\}$，あるいはHP出力$\{e_m^{(H)}\}$として，2サンプルにダウンサンプリングされているようす」

［式（53），式（54）］:「2サンプルの第1段目のLP出力$\{e_m^{(L)}\}$が，さらに第2段目のLP出力$\{e_m^{(LL)}\}$，あるいはHP出力$\{e_m^{(LH)}\}$として，1サンプルにダウンサンプリングされているようす」

〔図20.8〕合成フィルタバンクの信号計算の流れ（オクターブ分割の場合）



● 合成フィルタバンク

次に，合成フィルタバンク（IDCT値の計算に相当）のシステム構成を図20.8に示す．ここで，説明の便宜上，分析フィルタバンクの三つの周波数帯域に分割された出力信号$\{e_m^{(LL)}\}$，$\{e_m^{(LH)}\}$，$\{e_{2m-1}^{(H)},\ e_{2m}^{(H)}\}$をそれぞれ，次のように表すことにする．

$$\begin{cases} p_m^{(LL)} = e_m^{(LL)},\quad p_m^{(LH)} = e_m^{(LH)} \\ p_{2m}^{(H)} = e_{2m-1}^{(H)},\quad p_{2m+1}^{(H)} = e_{2m}^{(H)} \end{cases} \quad \cdots\cdots (55)$$

● 第1段目

$$p_{2m}^{(L)} = p_m^{(LL)} + p_m^{(LH)},\quad p_{2m+1}^{(L)} = p_m^{(LL)} - p_m^{(LH)} \quad \cdots (56)$$

● 第2段目

$$\begin{cases} y_{4m} = p_{2m}^{(L)} + p_{2m}^{(H)} \\ y_{4m+1} = p_{2m}^{(L)} - p_{2m}^{(H)} \\ y_{4m+2} = p_{2m+1}^{(L)} + p_{2m+1}^{(H)} \\ y_{4m+3} = p_{2m+1}^{(L)} - p_{2m+1}^{(H)} \end{cases} \quad \cdots\cdots (57)$$

ここで式（56），式（57）はIDCT値の計算に相当し，次のような処理を表している．

［式（56）］:「1サンプルの信号$\{p_m^{(LL)}\}$と$\{p_m^{(LH)}\}$から，第1段目のLP出力，あるいはHP出力として，2サンプルの信号$\{p_m^{(L)}\}$にアップサンプリングされているようす」

［式（57）］:「2サンプルの第1段目のLP出力$\{p_m^{(L)}\}$とHP出力$\{p_m^{(H)}\}$から，第2段目のLP出力，あるいはHP出力と

して，4サンプルの信号$\{x_{4m},\ x_{4m+1},\ x_{4m+2},\ x_{4m+3}\}$にアップサンプリングされているようす」

●“完全再構成条件”

まず，式(51)，式(52)を式(53)，式(54)に代入することにより，“分析フィルタバンク”の四つの周波数帯域に分割された出力は，4サンプルの信号$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$を用いて次のように表される．

$$e_m^{(LL)} = \frac{x_{4m} + x_{4m-1} + x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots\cdots\cdots (58)$$

$$e_m^{(LH)} = \frac{-x_{4m} - x_{4m-1} + x_{4m-2} + x_{4m-3}}{4} \quad \cdots\cdots\cdots\cdots (59)$$

さらに続けて，式(52)，式(58)，式(59)を式(55)～式(57)に代入することにより，“合成フィルタバンク”の4サンプルの出力信号$\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$が次のように導かれる．

●第1段目

$$\begin{cases} p_{2m}^{(L)} = \dfrac{x_{4m-2} + x_{4m-3}}{2}, \quad p_{2m+1}^{(L)} = \dfrac{x_{4m} + x_{4m-1}}{2} \\ p_{2m}^{(H)} = \dfrac{-x_{4m-2} + x_{4m-3}}{2}, \quad p_{2m+1}^{(H)} = \dfrac{-x_{4m} + x_{4m-1}}{2} \end{cases} \quad \cdots (60)$$

●第2段目

$$\begin{cases} y_{4m} = p_{2m}^{(L)} + p_{2m}^{(H)} = \dfrac{2x_{4m-3}}{2} = x_{4m-3} \\ y_{4m+1} = p_{2m}^{(L)} - p_{2m}^{(H)} = \dfrac{2x_{4m-2}}{2} = x_{4m-2} \\ y_{4m+2} = p_{2m+1}^{(L)} + p_{2m+1}^{(H)} = \dfrac{2x_{4m-1}}{2} = x_{4m-1} \\ y_{4m+3} = p_{2m+1}^{(L)} - p_{2m+1}^{(H)} = \dfrac{2x_{4m}}{2} = x_{4m} \end{cases} \quad \cdots\cdots (61)$$

以上より，式(51)～式(54)は“分析フィルタバンク”における周波数成分ごとに分割する信号処理，式(56)，式(57)は“合成フィルタバンク”における再合成する処理に相当することがわかる．よって，式(61)より，入力信号のブロック$\{x_{4m-3},\ x_{4m-2},\ x_{4m-1},\ x_{4m}\}$と出力信号のブロック$\{y_{4m},\ y_{4m+1},\ y_{4m+2},\ y_{4m+3}\}$との間には，$k = 4m,\ 4m+1,\ 4m+2,\ 4m+3$として，

$$y_k = x_{k-3} \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (62)$$

となる“完全再構成条件”が導かれる．

例題3

いま，ディジタル信号を，

$\{3,\ 3,\ 3,\ 3,\ 6,\ (-6),\ 6,\ (-6)\}$

とするとき，“オクターブ分割によるツリー構成(図20.3)”で処理したとき，DCT値，IDCT値を計算し，式(62)の“完全再構成条件”が成立することを検証せよ．

解答3

●DCT処理

分析フィルタバンクで処理すればよい．すなわち，式(51)～式(54)において$m = 0,\ 1$とし，4サンプルを1ブロックとして計算する．次に，計算の流れを示す．

$\{x_{-3}=3,\ x_{-2}=3,\ x_{-1}=3,\ x_0=3\}$

$\{x_1=6,\ x_2=-6,\ x_3=6,\ x_4=-6\}$



●第1段目

$$e_{-1}^{(L)} = \frac{x_{-2}+x_{-3}}{2} = 3,\quad e_0^{(L)} = \frac{x_0+x_{-1}}{2} = 3$$

$$e_1^{(L)} = \frac{x_2+x_1}{2} = 0,\quad e_2^{(L)} = \frac{x_4+x_3}{2} = 0$$

$$e_{-1}^{(H)} = \frac{-x_{-2}+x_{-3}}{2} = 0,\quad e_0^{(H)} = \frac{-x_0+x_{-1}}{2} = 0$$

$$e_1^{(H)} = \frac{-x_2+x_1}{2} = 6,\quad e_2^{(H)} = \frac{-x_4+x_3}{2} = 6$$

●第2段目

$$e_0^{(LL)} = \frac{e_0^{(L)}+e_{-1}^{(L)}}{2} = 3,\quad e_1^{(LL)} = \frac{e_2^{(L)}+e_1^{(L)}}{2} = 0$$

$$e_0^{(LH)} = \frac{-e_0^{(L)}+e_{-1}^{(L)}}{2} = 0,\quad e_1^{(LH)} = \frac{-e_2^{(L)}+e_1^{(L)}}{2} = 0$$

●IDCT処理

合成フィルタバンクで処理すればよい．すなわち，式(55)～式(57)において$m = 0,\ 1$とし，4サンプルを1ブロックとして計算する．次に，計算の流れを示す．

$\{p_0^{(LL)} = e_0^{(LL)} = 3,\ p_1^{(LL)} = e_1^{(LL)} = 0\}$

$\{p_0^{(LH)} = e_0^{(LH)} = 0,\ p_1^{(LH)} = e_1^{(LH)} = 0\}$

$\{p_0^{(H)} = e_{-1}^{(H)} = 0,\ p_1^{(H)} = e_0^{(H)} = 0,$

$p_2^{(H)} = e_1^{(H)} = 6,\ p_3^{(H)} = e_2^{(H)} = 6\}$



●第1段目

$p_0^{(L)} = p_0^{(LL)} + p_0^{(LH)} = 3,\ p_1^{(L)} = p_0^{(LL)} - p_0^{(LH)} = 3,$

$p_2^{(L)} = p_1^{(LL)} + p_1^{(LH)} = 0,\ p_3^{(L)} = p_1^{(LL)} - p_1^{(LH)} = 0$

●第2段目

$y_0 = p_0^{(L)} + p_0^{(H)} = 3,\ y_1 = p_0^{(L)} - p_0^{(H)} = 3,$

$y_2 = p_1^{(L)} + p_1^{(H)} = 3,\ y_3 = p_1^{(L)} - p_1^{(H)} = 3,$

$y_4 = p_2^{(L)} + p_2^{(H)} = 6,\ y_5 = p_2^{(L)} - p_2^{(H)} = -6,$

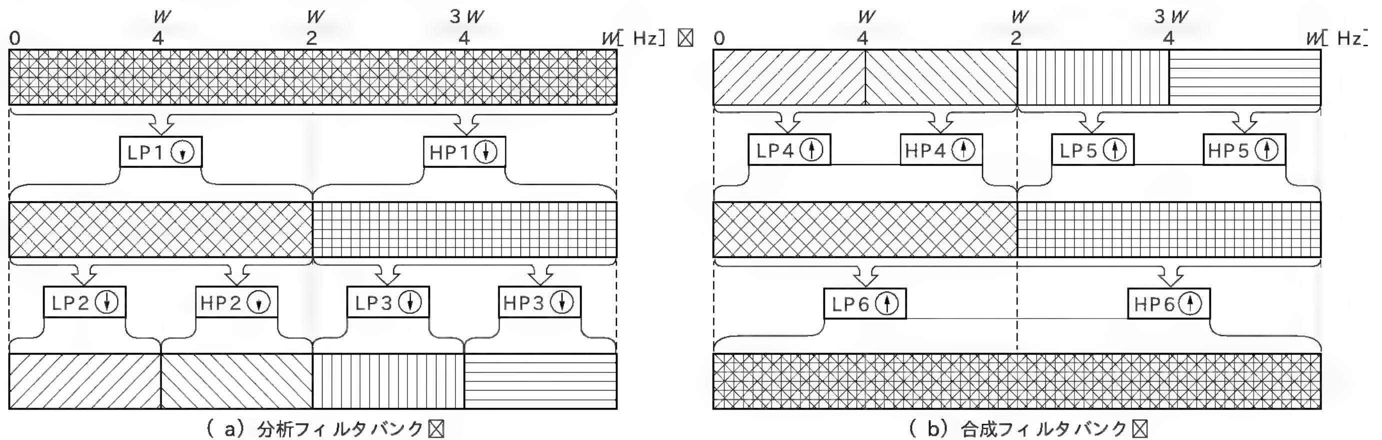
$y_6 = p_3^{(L)} + p_3^{(H)} = 6,\ y_7 = p_3^{(L)} - p_3^{(H)} = -6$

以上の計算結果より，$y_k = x_{k-3}$の関係が成立しているので，“完全再構成条件”が確認される．

ツリー構成における周波数帯域の分割特性

DCT/IDCT計算において，2分割以上の帯域分割を行い，サブバンド信号を得るシステムがツリー構成とよばれるものであり，等分割，オクターブ分割の2種類について，その分割特性の特徴を説明する．例として，4サンプルを1ブロックとするフィルタバンク構成を考える(図20.2，図20.3)．





[図20.9] 等分割による帯域特性(4分割の例)

(a) 分析フィルタバンク

(b) 合成フィルタバンク

● 等分割による帯域特性

図20.2のツリー構成において，四つの同じ帯域幅を有するように周波数分割するシステムである(図20.9)．

● 分析フィルタバンクにおける帯域分割

まず，周波数全体(0〜W[Hz])をローパスフィルタLP1とハイパスフィルタHP1でそれぞれ，

$$\left(0 \sim \frac{W}{2}\right), \left(\frac{W}{2} \sim W\right)$$

の帯域に大きく分割する．次に，LP1の出力の周波数成分，

$$\left(0 \sim \frac{W}{2}\right)$$

をローパスフィルタLP2とハイパスフィルタHP2でそれぞれ，

$$\left(0 \sim \frac{W}{4}\right), \left(\frac{W}{4} \sim \frac{W}{2}\right)$$

の帯域に細かく分割する．同様に，HP1の出力の周波数成分，すなわち，

$$\left(\frac{W}{2} \sim W\right)$$

をローパスフィルタLP3とハイパスフィルタHP3でそれぞれ，

$$\left(\frac{W}{2} \sim \frac{3W}{4}\right), \left(\frac{3W}{4} \sim W\right)$$

の帯域に細かく分割する．以上の帯域分割により，DCT値を計算する等分割の分析フィルタバンクでは，次の四つの周波数成分に等しい帯域幅に分けられることが理解される．

$$\left(0 \sim \frac{W}{4}\right), \left(\frac{W}{4} \sim \frac{W}{2}\right), \left(\frac{W}{2} \sim \frac{3W}{4}\right), \left(\frac{3W}{4} \sim W\right)$$

● 合成フィルタバンクにおける帯域分割

続けて，合成フィルタバンクでは分析フィルタバンクの逆のフィルタ操作により，IDCT値を計算する．つまり，

$$\left(0 \sim \frac{W}{4}\right), \left(\frac{W}{4} \sim \frac{W}{2}\right), \left(\frac{W}{2} \sim \frac{3W}{4}\right), \left(\frac{3W}{4} \sim W\right)$$

の四つに等分割された信号のうち，

$$\left(0 \sim \frac{W}{4}\right), \left(\frac{W}{4} \sim \frac{W}{2}\right)$$

の周波数成分の信号からLP4とHP4を用いて，

$$\left(0 \sim \frac{W}{2}\right)$$

の周波数成分の信号を再合成する．同様に，

$$\left(\frac{W}{2} \sim \frac{3W}{4}\right), \left(\frac{3W}{4} \sim W\right)$$

の周波数成分の信号からLP5とHP5を用いて，

$$\left(\frac{W}{2} \sim W\right)$$

の周波数成分の信号を再合成する．さらに，

$$\left(0 \sim \frac{W}{2}\right), \left(\frac{W}{2} \sim W\right)$$

の周波数成分の信号からLP6とHP6を用いて，周波数全体(0〜W[Hz])の信号を再合成する．

● オクターブ分割による帯域特性

図20.3のツリー構成において，三つの帯域に周波数分割するシステムである(図20.10)．等分割と違う点は，帯域幅が等しくないことである．

● 分析フィルタバンクにおける帯域分割

まず，周波数全体(0〜W[Hz])をローパスフィルタLP1とハイパスフィルタHP1でそれぞれ，

$$\left(0 \sim \frac{W}{2}\right), \left(\frac{W}{2} \sim W\right)$$

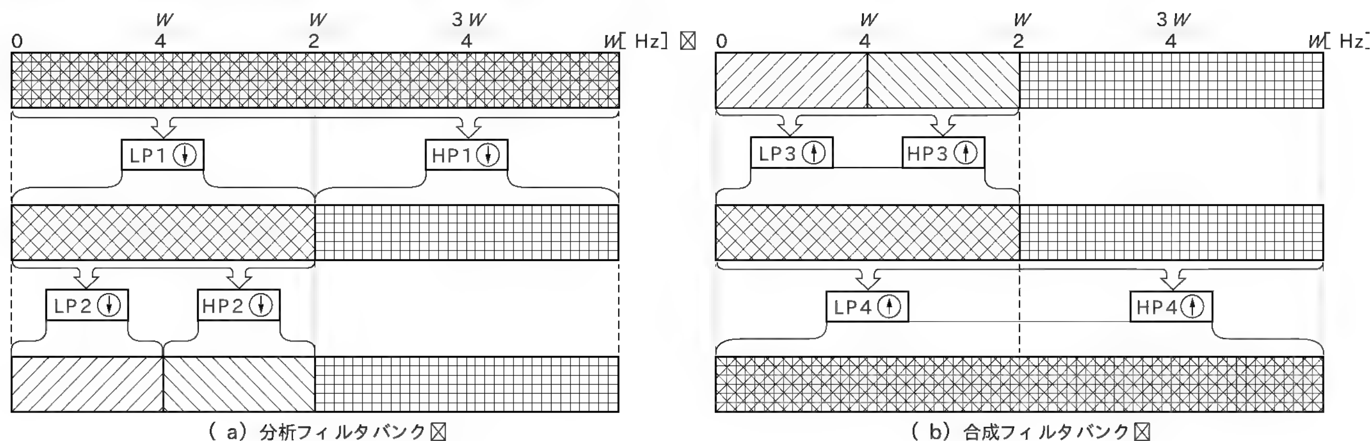
の帯域に大きく分割する．次に，HP1の出力の周波数成分，すなわち，

$$\left(\frac{W}{2} \sim W\right)$$

はそのままにして，LP1の出力の周波数成分，すなわち，

$$\left(0 \sim \frac{W}{2}\right)$$

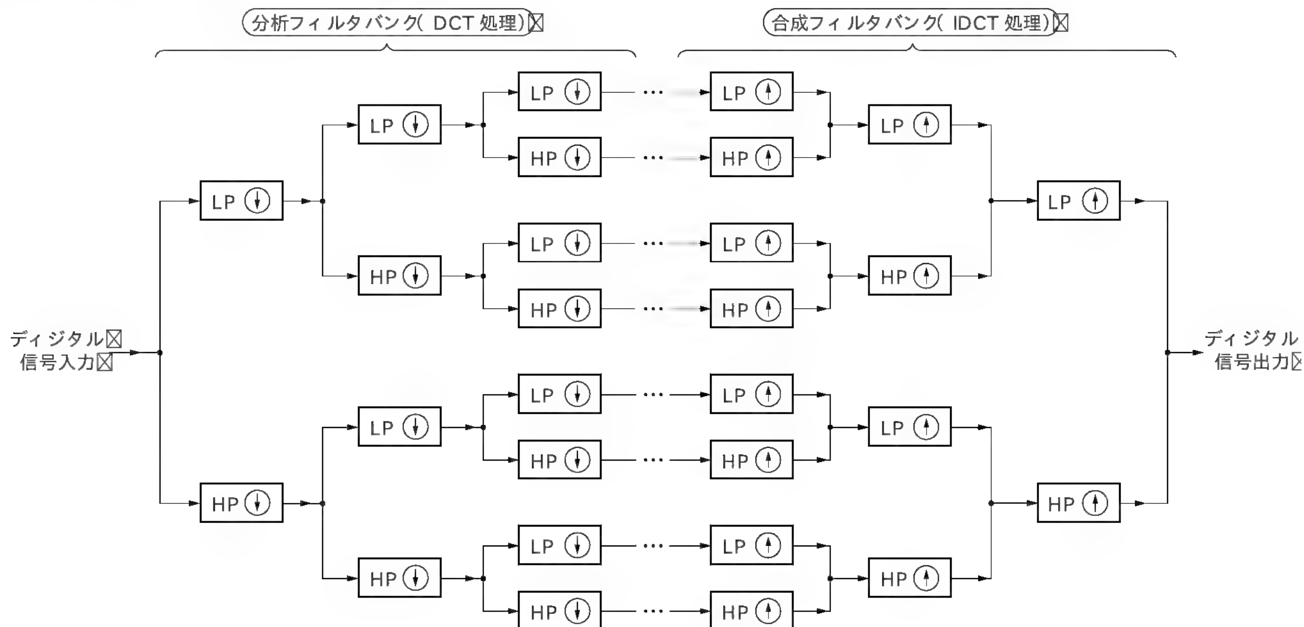
〔図20.10〕オクターブ分割による帯域特性(3分割の例)



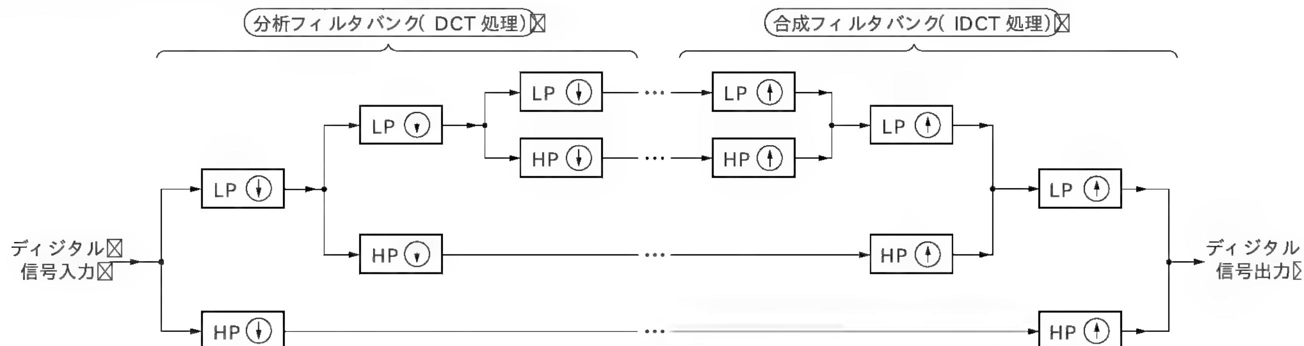
(a)分析フィルタバンク

(b)合成フィルタバンク

〔図20.11〕ツリー構成による等分割処理 例題4



〔図20.12〕ツリー構成によるオクターブ分割処理 例題4





のみをローパスフィルタ LP2とハイパスフィルタ HP2でそれぞれ，

$$\left(0\sim\frac{W}{4}\right),\ \left(\frac{W}{4}\sim\frac{W}{2}\right)$$

の帯域に細かく分割する．以上の帯域分割により，DCT 値を計算する分析フィルタバンクでは，

$$\left(0\sim\frac{W}{4}\right),\ \left(\frac{W}{4}\sim\frac{W}{2}\right),\ \left(\frac{W}{2}\sim W\right)$$

のように，帯域幅$W/4$と$W/2$の三つの周波数成分に分けられることが理解される．ここで，帯域幅$W/4$と$W/2$が2倍の関係にあることがオクターブ(octave)と呼ばれることから，図20.3のシステムは“オクターブ分割によるツリー構成”と称される．なお，“等分割によるツリー構成”との最大の相違点は，ハイパスフィルタの出力を細かく帯域分割するかどうかである．

● 合成フィルタバンクにおける帯域分割

続けて，合成フィルタバンクでは分析フィルタバンクの逆のフィルタ操作により，IDCT 値を計算する．つまり，

$$\left(0\sim\frac{W}{4}\right),\ \left(\frac{W}{4}\sim\frac{W}{2}\right),\ \left(\frac{W}{2}\sim W\right)$$

の三つに帯域分割された信号のうち，

$$\left(0\sim\frac{W}{4}\right),\ \left(\frac{W}{4}\sim\frac{W}{2}\right)$$

の周波数成分の信号から LP3と HP3を用いて，

$$\left(0\sim\frac{W}{2}\right)$$

の周波数成分の信号を再合成する．さらに，

$$\left(0\sim\frac{W}{2}\right),\ \left(\frac{W}{2}\sim W\right)$$

の周波数成分の信号から LP4と HP4を用いて，周波数全体(0～W[Hz])の信号を再合成する．なお，このオクターブ分割によるツリー構成は，“ウェーブレット変換”の考え方に直接関係あることを記憶にとどめておいてもらいたい．

例題4

いま，8サンプルを1ブロックとみなして，帯域分割してツリー構成(等分割，オクターブ分割)による DCT/IDCT 計算するシステムを示せ．

解答4

上述の等分割，オクターブ分割のいずれも，ツリー構成は，一般的な分割に容易に拡張できることは明らかである．なお，図20.11と図20.12に2種類のシステム構成を示しておくので，DCT/IDCT 計算における周波数帯域の分割特性を考えてみてほしい．

次回は，DCT の高速計算アルゴリズムと応用例を採り上げ，わかりやすく解説していく予定である．お楽しみに．

みたに・まさあき　東京電機大学工学部情報通信工学科

シニアエンジニアの技術草子 参拾五之段

◆堕ちた偶像

旭 征佑

● 不吉な前兆？

筆者が自宅で使っているパソコンはCeleronの800MHzにWindows 2000，そしてOfficeXPである．古い仕様だと思われるかもしれないが，メモリをフルに搭載し，2年近くの間，至って快適に動いているのでとくに買い換える気はなかった．ところがある日，ハードディスクに異常が発生した．ほかにもパソコンはあるが，この際だから，もっと高速な新しいパソコンを用意することにした．

こんなときは，いつもなら自作するのだが，今は忙しい時期だ．そんなへ理屈をつけ，しばらくぶりにメジャーな日本製のパソコンを買うことにした．

たまたまショップに行く用事のあった同僚と一緒にショップに出かけ，早速買うことに決めたのが，DVD-RAMドライブ，TVチューナ搭載と，筆者には不似合いのノートパソコンだった．同僚には，「似合わない」とか，「DELLにすれば半額で済む」などと悪態をつかれ，一瞬購入をためらったが，「えいっ」とばかりに勢いで買ってしまった．仕様もCPUは同じCeleronだが2.0GHzだし，ハードディスクも80Gバイトあり，しばらくは使えそうだ．TVも見られるし，DVD-RAMに録画もできる．ちょっと楽しみだ．しかし，これが，それから1週間以上にわたって苦しむ原因になるとは，誰が予想できただろうか．

● 悪戦苦闘の始まり

持ち帰って早速電源を入れた．第一印象として，15インチの液晶画面はショップでみるよりずっと大きくて明るかった．コントラストの強さには最近の技術の片鱗を感じたが，筆者のように長時間パソコンと付き合うには，少し目が疲れるかもしれない．しかし，そんなことより，ずっと気になることが次々と発生した．

まずは，箱の中に入っているソフトウェアのユーザー登録はがきらしきもの，プロバイダ申し込みのはがき，そのほかのソフトウェアのチラシなどが山のように入っている．こんなにいっぱいあったのでは，どれが必要な書類なのかわからないではないか．最初に適当に判断し，大部分をごみ箱行きにした．

パソコンに名前やシリアルなどを登録すると，すぐにWindows XPが立ち上がるようになった．次に気がついたのは，今まで使っていたCeleron 800のマシンより圧倒的に遅いということだ．残念だが，XPでノート用のハードディスクだとこの程度かもしれない，などと勝手に納得することにした．若干の悔いが残る．

クイックシートというビギナー用のシートがついていて，「アプリケーションのはじめ方」では，スタートボタンを押しなさいと書いてある．ほかにも，「文字入力の仕方」，「ローマ字のつづり方」，「ホームページを見るには」……といった簡単な案内があるが，買い替えユーザーが増えてきている現在，あまり使い物にならないだろう．第一，このレベルの人は，この程度の解説で納得するとも思えない．

さて，インターネットにつなごうと思うのだが，アンチウィルスソフトが見当たらない．3か月有効なバージョンが標準で付いてくるはずだが…．通常，マニュアルなど読まずにセッティングしてしまう性格なのだが，今回ばかりはマニュアルを読まざるを得なかった．そうして，3冊あるマニュアルの2冊目を読んでいる途中に，メーカー独自のツールからインストールする必要があることがわかった．ついでに，地図ソフトもインストールした（余計なことをしてしまった）．

マニュアルも，最初は各部の名称などの説明があるが，次が，「メモリを増設する」とか「TVを見る」というページだ．妙な構成だと思う人が多いかもしれない．

「TVを見る」というページには，ケーブルの接続の方法までは書いてあるが，そのあとの操作方法は，「ヘルプをご覧ください」とあり，そのヘルプには変な固有名詞がついていてよくわからない（あとで，メーカーのサポートページの固有名称だとわかった．マニュアルを飛ばし読みしていたからわからなかったのだ）．メモリを増設するというページについては，何もこんなこと，最初に説明しなくてもいいのに，などと思った．

そんなこんなでバタバタしていると，先の同僚から意地悪なメールが来た．「メモリを増設しないと，遅くて使い物にならないよ」．今ごろいわれても…．しょうがない．再びショップに行って512Mバイトのメモリを買ってきて追加した．すると今度は見違えるほど速くなった．なんだ，こんな簡単なことだったのか．マニュアルの最初のほうにメモリの増設方法が書いてある理由がこれでやっとわかった．

近くのスーパーでDVD-Rディスクを買ってストックしておこうと思い，ドライブが何倍速に対応しているのかを調べたが，



シニアエンジニアの技術草子

マニュアルやWebサイトには，ドライブの細かい仕様が書かれていない．ライティングソフトを起動して，ドライブ情報を得ることで，やっと2倍速対応だと知ることができたが，これでは本末転倒ではないか．

そうして，実際に使い出すと，またこれが使いにくい．Adobeのソフトをいくつかインストールしたら，スタートボタンを開いて出てくるメニューが3列になった．メニューの数にして70を超える．筆者はソフトの体験版なんていらないし，家計簿やレシピ，医学辞典もいらない．逆にこんなにあると，スタートボタンから必要なアプリケーションを探すのが億劫になるだけだ．できれば，オンデマンドローディングにしてほしい．

こんな感じで，気になる点がなくなるまで1週間近くかかった．メーカー製のパソコンは簡単に設定できるものとタカをくくっていたが，この新しいノートパソコンにかなりの時間を取られていた．これだったら，自作するか，DELLを買っておいたほうが圧倒的によかった．

● 日本製パソコンは，いつからこうなったのか？

「設定が簡単で，すぐに使えるはずだ」という考えで日本製のパソコンを選んだのは，じつは大きな誤りだった．1社だけのパソコンを見て日本製のパソコンを批判するのはいいすぎかもしれないが，筆者の知る限りどこも似たようなものだ．一般消費者向けに販売されているメーカーのパソコンは，非常に扱いにくい．メーカーは，買った客が本当に，何に困っているのかを調査したことがあるのだろうか？

個別指導の小さなパソコンスクールを経営している友人に聞いてみた．一般の人は，メーカーのブランドでパソコンの購入を決めている．しかし，パソコンの設定はほとんど何もしていない．付いているアプリケーションについては，ほとんどの人がまったく使用していないし，何が入っているのかも知らない．これが「パソコンがわからない」という理由の一つになっているという．

デスクトップに表示されている宣伝などのアイコンは，意味が理解できず，しかも消し去る手段を知らない．消す方法は，マニュアルには記載されていないか，記載されていても非常にわかりにくい．

● 日本のパソコンメーカに期待すること

製品の開発は，技術の動向，各メーカーの開発動向，マーケットの動向の三つを見極める必要がある．思うに，最後のマーケット動向は，この場合，ターゲットのユーザー動向だろう．



何をされてもメーカー志向が強いのは，高度成長以来培われた日本人の特性かもしれない．しかし，最近の若い人はメーカーに依存しない．その傾向はDELLの大躍進にも表われている．

日本メーカーのパソコンは，以前はとても信頼できた．今や年間千数万台も売れるパソコンではあるが，その日本メーカーがユーザーの側を見ているとは到底思えない．ユーザーの動向を見誤り，過去の成功体験に基づいて自己の路線を歩み続けたメーカーは力尽きてきたことを忘れたのか．

以前，モデルチェンジしないシンプルなパソコンを出してほしいと書いたことがある．このような状況では，すぐに日本メーカーのパソコンの大きな革新は望めそうもない．贅沢はいわない，せめて，箱を開いたら30分で必要な設定が終えられる，そんなパソコンを出してほしい．そうしないと，ユーザーからソッポを向かれるのは，そんなに遠くない未来のような気がする．

あさひ・しょうすけ　テクニカルライター
イラスト　森　裕子

Engineering Life in

エンジニア達の健康管理・健康への努力(第二部)

前回はアメリカの太りやすい食生活について説明したが，今回は健康管理について説明する．食生活以外に日米の格差があり，これらが健康管理への意識にも影響していると思う．

☆ 医療費が高い＝健康管理は自己責任

まず大きな違いとして挙げられるのは，医療保険の違いではないだろうか．アメリカでは，全般的に日本よりも医療代が高い．国民保険などのように国家レベルで医療価格が決められていないので，基本的には民間の健康保険に入ることが必須となる．

しかし，個人で入ると思いのほか高いので会社を通じて入るのがごく一般的だ．アメリカ人は，会社の福利厚生は給与以外での報酬を評価するポイントとして捉えている．たとえば，歯科保険の中に一般的な保険では含まれていない矯正や歯の美白[注1]が入っているかどうかで大きな差が出る．

こんな話もある．二つの会社から転職のオファーがあった際に，両方とも似たような報酬だが，一方に矯正や歯の美白が入った盛りだくさんの歯科保険が付くと，そちらに傾く場合がある．その他には，眼科の保険があり，毎年検眼があって新しいメガネのフレームをどれぐらい保険でカバーできるかということも，毎日端末に向かって仕事をしているエンジニアにとっては大切なことだ．また，とくに子供をもつ家族にとっては医療費が心配になるので，会社を選ぶ大きな要件になることが多い．しかし，レイオフなどもあるので，いつまでも会社にいて健康保険があるともいいがたいし，老後もシリコンバレーの会社であったような健康保険に入っていられるという保証もない．

いずれにせよ，保険があっても医療費が高いので自己負担額などを考えるとやはり自分で健康管理をしないといけないというのが一般的な認識だ．そこでエンジニア達は自分の健康を考えて行動を取ることになる．その一方で，太りやすい物を大量に食べる点などで矛盾も感じるかもしれないが，長期的に自分の健康を考えると禁煙[注2]するとか，食べる量を減らすとか，運動をするといったことを真剣に考えるのだと思う．

☆ 医療以外のケアにも関心あり

医療に関係しては，普通の医者に通う以外にも，Alternative Medicineと一般に呼ばれる，通常医療でない方法で健康になろうというのも関心が高い．カイロプラクティック，中国医学(針や漢方薬)，指圧，マッサージがこの種類に入り，普通の医者に通っても治らない症状を見てもらうものだ．一般的に腰痛とかアレルギなどで悩む人が多い．

そのほかに心理的な部分での健康も関心が高く，会社でまとめてセラピーを受けられる健康保険に入っている会社が増えているし，専属のセラピストと契約している会社もある．アメリカではセラピストが一般化しており，学校でもだいたい専属がかならずいる．おもしろいところでは役員会や幹部会に出席するセラピストやメンタルケアプロがいる．この人達は幹部会議で険悪なムードになったり個人的攻撃に議論がエスカレートしないように仲裁[注3]に入ったり，倫理面やドロドロとした人間関係に関してアドバイスするそうだ．

ヨガ，太極拳，そして禅[注4]を組むことなども多くのエンジニア達に支持を得ている．いずれも内面的な「健康」を保つ手段として，プレッシャが多くて勤務時間の長いエンジニア達には有意義な方法であると考えられる．

☆ 自動車通勤の影響

次に大きな違いは，自動車での移動ではないだろうか．シリコンバレーでは公共交通網がほとんど整備されていないので車が必須となる．自動車通勤がごく一般的なので，日本の都心部の満員電車で通勤するような状況ではない．エンジニアだと，どうしても座った一日が多くなるのは日米間で格差はないと思うが，しかし生活パターンがかなり異なっていると思う．まず駅まで歩いたり，会社まで歩くことがほとんどない．また通勤以外にも車での移動がほとんどなので，意図的に「歩こう」と思わない限り歩くことはない．

通勤が車なのでついつい朝食は車の中で…コーヒーに何か片手で食べられる物を持ち込んで通勤…というエンジニア達も多い．

またこれに関連して，飲酒運転[注5]は厳しく取り締まられるので，仕事の後にフラッと気軽に飲みに行くということがなかなかできなくなる．自動車のおかげでほとんど座ったままの生活をしているエンジニア達が多いのはたしかだ．多くの会社には社員の運動を支援するためにシャワールーム[注6]を備えた会社が多い．ここでさっぱりしてから一日のをスタートするわけだ．

☆ 健康食オタク

食事は豊富で太りやすい物が多いし，車の移動が多く，医療費も高いので健康管理をするには自分から行動を起こさなければならない．そこで，食べることにこだわるのも一つの方法であるとエンジニア達は考える．無農薬，有機農法の野菜などを取り扱う健康食品店や高級スーパーマーケットがあり，普通のスーパーに比べると値段はかなり高いが，健康にこだわるエンジニア達には人気だ．食品以外にサプリメント類やハーブや薬

注1：矯正や美白にはかなり関心が高い．

注2：アメリカも禁煙している人は多く，シリコンバレーでも喫煙してる人は少ない．

注3：筆者が以前勤めていた会社で本当にあった．

注4：大型の禅道場が存在するし，アップルの会長のSteve Jobs氏は若いころから禅に高い関心を持っていたことで有名だ．ほかの瞑想方法にも人気がある．

注5：日本から出向・駐在エンジニアで飲酒運転のトラブルが多い．

注6：以前もこのコラムで紹介したが，シャワールームは会社に合宿するエンジニアのロッカールームと化してしまうことも多い．

草(漢方薬系が多い)も豊富に揃えている．病気になる前に身体に良いものを…と考えて健康食品やサプリメントにこだわる「健康オタク」達だ．

個人で，ある程度のこだわりがあるかと思うが，極端な場合は菜食主義[注7]になって肉や魚を一切食べないとか，野菜は無農薬のものしか食べない…というエンジニアもいれば，ジャンクなドーナツを食べながらサプリメントはきっちりと飲んでいるという矛盾した人達もいる．

☆ いたるところにあるジム

もっとも，食べ過ぎなので食べる量を減らすとか，揚げ物とか脂っこい物を減らすとか，そういう努力もしているエンジニア達もいる．また一度ついた脂肪を落とすため，または継続的に健康管理をするためにも運動を考える人達も多い．シリコンバレーにはフィットネスジムがかなりの数があり，夕方からけっこう混み合うことが多い．

大きな会社になるとビル内にジム施設を持っていて，会社の福利厚生の一部として自慢の種になる．または，会社でジムのメンバーシップを購入している例もある．住んでいる賃貸マンションやアパートにジムがある場合もあり，サービスの差別化として提供されている．まあ，一般のエンジニアは大体が自腹で大型ジムのメンバーシップを払って入るが，普通のジムだと月々30～50ドル程度なのでそれほど高いとは思われない．

「高級」に分類されるジムもシリコンバレー近辺に2～3軒あり，これらではゴルフの会員権に似たメンバーシップを買い，さらに月々の使用料が発生するので，なかなかのお値段らしい．高級なジムには運動以外に綺麗なレストランやラウンジ(クラブハウス)があったり，しっかりとした託児所機能が付いている．運動の内容はジムによって違ってくるが，大体はウエイトトレーニング(マシンまたはフリーウエイト)，ランニングマシンやエアロバイクなどのエアロビクスマシン，インストラクタがリードするグループエクササイズ(エアロビクス，キックボクシング，ヨガ，スピニングなど)があり，それ以外にプールがあったり，バスケットボールのコートがあったり，ラケットボールやスカッシュがある．そのほかに岩登りのトレーング用のインドアクライマに特化したジムやヨガ専門の道場なども人気だ．

年齢的にもさまざまな人が利用するし，運動レベルもまちまちだ．つまり，凄くマッチョなエンジニアもいれば，最近ジムに入会したばかりの人達など，いろいろなレベルやタイプの人が利用する．日本でも定着しつつあると思うが，夕方に行くと50台近くあるエアロビクスマシンが全部使われていて，皆が黙々と汗をかいているという，ちょっと異様な雰囲気もある．一般論で申し訳ないが，アメリカでは中学校からウェイトマシンを使うことがあるので，とくに男性がマシンエクササイズをすることにあまり違和感はない．

もっとも，ステロタイプ的なひょろっとして色白でナヨナヨしいエンジニアは嫌だ…という人が多いので自らマッチョを目ざしてウエイトトレーニングに励むことも多い．女性はやはり筋肉が急激に付くのが怖い…というのでグループエクササイズに傾く方向が多いが，ウエイトをかなりやっていて，びっくりするほど引き締まった人達も多くいる．

☆ 忙しいエンジニアにはぴったり！　一石三鳥なジム通い

ジムに来る目的は人によりさまざまであるが，いくつかの複合的な目的があるように思える．シリコンバレーははっきりいってナイトライフや，おもしろい歓楽街があるわけでもないので，ジムがある種の社交の場や娯楽でもあるとも思える．24時間営業しているジムもあれば，朝の5時から夜の12時まで空いている大型ジムがあり，仕事の合い間に汗をかいたり，職場以外の人達と知り合うには良い場所なのかもしれない．

なんでもSun Microsystems社のScott McNealy氏はかなりのスポーツマンらしいが，朝の6時からジムで3対3のバスケットを幹部や取引先の幹部達とやって汗を流してから会社に行くそうだ．大まかに，会社に行く前に運動する人達，昼休みに昼食を食べないで運動する人達，そして仕事が終ってから来る人達に分けられる．決まった日程でジムに通うとそのうち顔見知りになる人が増えていくわけで，仲間や友達も増えていく．夜の部になるとやはり社交性が高いように感じる．女性もしっかりと化粧をしてジムに来るし，運動をそこそこにして女性にやたらに話しかけるエンジニア達も多い．もっとも，皆がナンパをしにジムに来ているというわけでもないが．

単純に運動をしてリフレッシュするという意味も大きいと思う．仕事中に気分転換に来て，仕事に戻るエンジニア達も多い．また大量に本を持ち込んで黙々とエアロビクスマシン(ランニングマシン，エアロバイクなど)にセットされている専用の書棚に本や新聞を置いてゆっくりと運動をしながら本を読んでいるエンジニア達も見かける．趣味の本やら新聞[注8]などなら少しは理解できるとして，会社のドキュメントやマニュアル類，教科書を読んでいるエンジニア達を見かけると何か複雑な気持ちになる．忙しいエンジニア達にとってジムに行くことは一石三鳥ぐらいなのだろうか？

トニー・チン　htchin@attglobal.net　WinHawk Consulting

注7：シリコンバレーでは，インド人などヒンズー教の人達は菜食が多いので，菜食レストランやメニューが一般化している．

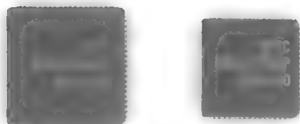
注8：ニューヨークタイムズなどの大型新聞になると読むのに1時間以上はかかる．

●16ビットマイコン

M16C/28グループ

- CPUコアは，16ビットの「M16C/60」を搭載．動作電圧は3.0～5.5V，最大動作周波数は20MHz，最小命令実行時間は50ns．
- 三相モータ制御回路の機能として，インプットキャプチャ/アウトプットコンペア機能を備えた16ビットタイマと，シングルシャント制御回路機能を内蔵．
- ハードウェアマルチマスタI²Cバス機能や，内蔵リング発振器の周波数変更機能，リセット後のリング発振器からの立ち上げ機能などを備えている．
- パッケージは，80ピンLQFP（12mm×12mm）および64ピンLQFP（10mm×10mm）の2種類を用意している．

●サンプル価格：¥800～¥900



■（株）ルネサス テクノロジ
TEL：03-5201-5279

●マルチCPU搭載LSI

MB87Q1100

- 二つのARM9 CPUコア（ARM926EJ-S/ARM946E-S）を内蔵したシステム評価用のLSI．
- 二つのCPUを同時に動作させて，それぞれ個別の処理を行うデュアルCPUモードと，片方のCPUだけを動作させるシングルCPUモードの両方でシステムの評価が可能．
- AHBのバス帯域幅を増やすことで，複数のマスタ（システムを制御するモジュール）から出される命令の同時処理を可能にし，システム全体の処理能力を向上．
- ユーザー独自のAHB対応IPとLSIを接続するAHB-Lite外部拡張機能を搭載しているため，さまざまなIPを含んだシステムの評価が可能．
- さまざまなシステム評価を効率的に行うことで，システムLSIの設計をより短時間で行えるようにする．
- T-Engineボードに搭載しての販売を予定．

●サンプル価格：¥25,000

■富士通（株）
TEL：03-5322-3321
E-mail：edevice@fujitsu.com

●ICカード用32ビットマイコン

AE57C

- CPU演算器および内部バス幅が32ビットの「AE-5」を採用．
- 最新のAES暗号処理や，DES暗号処理，処理すべき乗剰余演算処理を実行するコプロセッサを搭載．
- 最大132KバイトのEEPROM，320Kバイトの大容量マスクROMを搭載しているため，アプリケーションの複数搭載，大容量データの格納が可能となり，多機能なICカードに対応可能．
- 周辺回路として，BEMやDMAC，電圧・周波数などの各種異常検出器，ウォッチドッグタイマ，乱数発生器などを備えている．

●サンプル価格：
¥1,700（ウェハ）
¥1,800（COT：Chip On Tape）

AE57C

■（株）ルネサス テクノロジ
TEL：03-5201-5238

●128MビットSDRAM

EDS1216AA EDS1216CA

- 0.11μmプロセスを採用した128Mビット（8Mワード×16）SDRAM．
- ディジタルビデオカメラ，DVDレコーダ，ディジタルテレビ，プリンタなどのディジタルコンシューマ機器向け．
- 電源電圧は，3.3V/2.5V品の二つを用意．
- セルフリフレッシュ時の消費電流は1.5mA，ローパワー品は0.6mA．
- ディジタルコンシューマ機器への単独接続を考慮し，出力ドライバ強度（Driver Strength）を「Half」，「Quarter」に設定可能．
- オーバシュート，アンダシュートノイズの低減を図っている．
- 実装面積を減らせる54ピンFBGA，従来品と置き換え可能な54ピンTSOP IIの2パッケージを用意．
- セットの小型化に貢献するMCP，SiPに対応するため，ベアチップでの出荷も予定している．

●サンプル価格：¥540（EDS1216AA）
¥820（EDS1216AC）

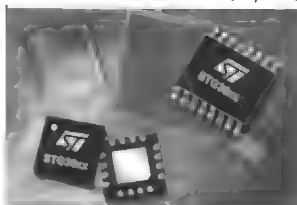
■エルピーダメモリ（株）
TEL：03-3281-1648

●小型アナログスイッチ

STG3699 STG3684

- STG3699はクワッドSPDTアナログスイッチ，STG3684はデュアルSPDTアナログスイッチ．
- スイッチのオン抵抗が0.5Ω未満であるため，音声回路の歪みは最小限に減少し，携帯電話などの製品の高品質なシグナルルーティングを保証．
- 複数の単極双投（Single Pole Dual Through）セルを持ち，1.8Vのコンパチブルディジタルコントロールピンを使用しているため，双方向モードでの動作が可能．
- スイッチは3×3mmの大きさのQFNパッケージに収納されており，省スペース化が図れる．

●サンプル価格：STG3699 ¥130（1,000個時）
STG3684 ¥120（1,000個時）



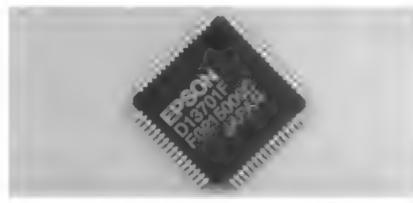
■STマイクロエレクトロニクス（株）
TEL：03-5783-8240 FAX：03-5783-8216

●有機ELディスプレイコントロールLSI

S1D13701

- 既存の表示コントロールLSIシリーズの技術をベースに，表示制御部を有機ELディスプレイに最適化．
- 有機ELディスプレイに直結可能な5.0Vフルスイング信号の出力機能を装備．
- ターゲットの表示サイズはカーオーディオに適する256ドット×64ライン，4bpp/16階調．
- 画像表示という大きな負荷を受け持つため，システムCPUの負荷を大幅に軽減することが可能．
- 24Kバイトの内蔵メモリを搭載．
- インテル80系/モトローラ68系のインダイレクトCPUインターフェースを装備．

●サンプル価格：¥1,000



■セイコーエプソン（株）
TEL：042-587-5816
URL：http://www.epsondevice.com/

●リアルタイムクロックLSI

M41ST85MX6

- シリアルRTCにマイクロプロセッサとNVRAMスーパバイザ機能を組み合わせ，32kHz水晶発振子を内蔵．
- シリアルRTC（リアルタイムクロック）とスーパバイザ機器を組み合わせた主電源とバックアップ電源の切り替え用LSI．
- 2.4mm×0.42×18.4mmのSoXパッケージ．
- 干渉から水晶発振子を隔離することでセキュリティを強化し，湿気に対する水晶発振子の保護機能を高めている．
- 水晶発振子の外部接続が不要となるため，アセンブリコストを軽減できる．
- プログラマブルでバッテリバックアップされるRTCに，1/100秒～100年までの分解能で時刻と日付けのRTCカウンタを格納．

●価格：下記へ問い合わせ



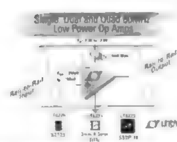
■ STマイクロエレクトロニクス（株）
TEL：03-5783-8240　FAX：03-5783-8216

●OPアンプ

LT6220 LT6221 LT6222

- LT6220はシングル，LT6221はデュアル，LT6222はクワッドレールトゥレール入出力．
- 60MHzで動作中のOPアンプは消費電流がチャネル当り1mA以下で，広帯域幅バッテリ駆動システムの動作時間を延ばすことが可能．
- 2.3V～12.6Vの広い電源電圧範囲で動作可能で，レールトゥレール入出力によって全電源電圧範囲を使用可能．
- 優れたDC精度を達成し，最大入力オフセット電圧が350μV，最大入力バイアス電流が150nA，CMRRが82dB，電圧利得が90dB．
- 50mAの出力電流を供給できるため，1mA/アンプ以下の低消費電流でもより大きな負荷をドライブすることが可能．

●サンプル価格：LT6220 ¥135～（1,000個時）
LT6221 ¥210～（1,000個時）
LT6222 ¥330～（1,000個時）



■ リニアテクノロジー（株）
TEL：03-5226-7291　FAX：03-5226-0268

●16ビットA-Dコンバータ

AD7621

- 3Mspsで動作し，INL（積分非直線性）は±1LSB，DNL（微分非直線性）は±1LSB．
- ミッシングコードが発生しない．
- 16ビット精度を提供するため，スキャナアプリケーションで鮮明な画像処理が可能となる．
- 三つの異なる変換レートモードを備えており，個々のアプリケーションに合わせて性能の最適化が図れる．
- 消費電力は100mW（typ.）と，低消費電力を実現．
- 5V単一電源で動作し，5Vもしくは3.3Vのディジタルロジックとのインターフェースが可能．
- 16ビット分解能を提供し，90dBのS/N比で18ビット800Ksps *SAR* A-Dコンバータ「AD7674」とピン互換．
- 内蔵変換クロック，内蔵リファレンスバッファ，エラー補正回路，シリアルおよびパラレルのインターフェースボードを装備．

●サンプル価格：$29.95（1,000個時）

■ アナログ・デバイセズ（株）
TEL：03-5402-8268

●オーディオサブシステム

LM4857

- アンプ，音量/ミキシング調整および3Dサウンドの機能を小型micro SMDパッケージに統合した携帯電話用Boomerオーディオサブシステム．
- チャネル当り495mW出力のハンドセット用ステレオスピーカドライバ，32段階音量調整と左右独立のステレオおよびモノ音量調整が可能な33mWステレオヘッドホンドライバを内蔵．
- 3.3V電源で動作し，高音質のステレオスピーカアンプ，43mWの電力を32Ω負荷に与えるモノイヤホンアンプ，外部電源方式ハンズフリースピーカ用ライン出力などの機能を装備．
- 3Dエンハンスト機能は，左右のスピーカが接近しすぎた際のステレオチャネル分解機能を改善し，システムサイズや機器の制約を解消．
- I^2C互換インターフェースを通じてコントロールされる．

●価格：¥324（1,000個時）

■ ナショナルセミコンダクタージャパン（株）
TEL：0120-666-116

●Webサーバ

ProDigio

- 周辺インターフェースとLAN，インターネット，携帯電話などのネットワークインターフェースが，モジュール内で稼動する最小クラスのWebサーバ．
- 搭載されている専用スクリプト言語を利用することで，標準でサポートするすべてのデバイスにアクセスができる．WebサーバやFTPなどのアプリケーションと連携することが可能．
- ファームウェア，ミドルウェア，接続に必要なドライバなどのソフトウェアを搭載．
- Webサーバ機能が内蔵されているため，接続された周辺機器，デバイスはWebブラウザより簡単にアクセスできる．
- CF-I/Fを2チャネル搭載し，無線LANカード，P-in m@ster，Air-H，MobileArkをサポート．

●価格：下記へ問い合わせ



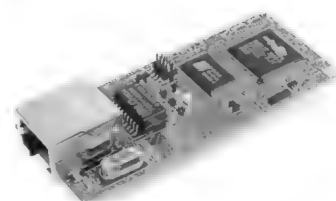
■（株）ティアンドデイ
TEL：0263-27-2131　FAX：0263-26-4281

●Ethernet用コアモジュール

RCM3700 Rabbit Core

- Rabbit3000マイクロプロセッサを搭載したコアモジュール．
- 512Kフラッシュ/512K SRAMまたは，256Kフラッシュ/128K SRAM，4本のシリアルポート，最小のフットプリント（75×30mm）を装備．
- デュアル低IDCヘッダによって，あらゆる種類のCMOSコンパチブルなディジタルデバイスと接続が可能．
- ディジタルI/O，電源，そのほかのシグナルは直接マザーボードに接続される．
- ＋5VDC耐性I/O，PWM出力，パルスキャプチャ，計測機能を装備．

●価格：¥4,800（100個時）



■ 東京電子販売（株）
TEL：03-5350-6711　FAX：03-5350-6867

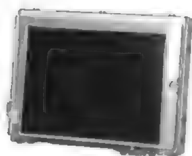
■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

●組み込み用カラーGUI端末

TS-05S TS-10S

• TS-05SはC/P重視の5.7型STNペアシャーシタイプ，TS-10Sは表示性，操作性重視の10.4型TFTフロントベゼルタイプ．
• ハードウェア（CPUボード，LCD，タッチパネル）およびミドルウェア（組み込みOS，ウィンドウシステム，シリアルアプリケーション）で構成され，RS-232-Cからのコマンドを介して，表示，操作の制御が可能．
• メンブレン（シート状）スイッチやメカニカルなキースイッチにより構成されていた操作パネルを，グラフィカルなスイッチやラベル表示に機能アップすることが可能．
• VDS独自のミドルウェア「Embedded View」の搭載により，PCのGUI環境に迫るオブジェクト表示が可能．

●価格：オープン価格



■ 日本ビクター（株）
TEL：046-278-1712 FAX：046-278-1757

●ディジタルパネルメータ

形K3HB-X 電圧・電流パネルメータ 形K3HB-V ロードセル，mvメータ 形K3HB-H 温度パネルメータ

• 生産現場の生産機器/設備，監視設備，検査ラインにおける表示器，計測判別器，警報器など幅広い用途に対応するインターフェース機器．
• 20ms（50回/s）と従来比3倍以上（温度入力タイプでは12.5倍）の高速サンプリングを実現．
• 判定出力，伝送出力の応答性の向上と平均化処理により，測定安定性を向上．
• 表示部にバックライト2色LED（赤/緑）付きネガLCD表示を採用し，判定動作出力に連動して計測表示部を緑色→赤色と変化させ状態が把握しやすい．

●価格：
¥31,500～¥55,000（電圧・電流パネルメータ）
¥38,000～¥60,000（ロードセル，mvメータ）
¥31,500～¥53,500（温度パネルメータ）



■ オムロン（株）
TEL：075-344-7080

●データ収集ボード

Multifunction I/Oボード Xシリーズ

• プラグインモジュールとシグナルコンディショニングプラットホーム，データ収集ボードをビルドアップ式にしている．
• プラグインモジュールは，数十種類のセンサに対応する各種センサアンプドライバユニットなどで，多数のラインナップがあり，複数，異種構成でプラットホームに搭載可能．
• プラットホーム上にリモートI/Oを搭載できるため，Ethernetケーブルを直結することができる．
• ソフトウェアは，目的とするセンサに合わせて，スケーリング，PID（フィードバック制御），リニアライズ，アラーム，データロガー，校正などを設定するだけで利用可能．

●予定価格：¥90,000～¥150,000

■ （株）サヤ
TEL：047-393-6136 FAX：047-393-6126
URL：http://www.saya-net.com/

●通信技術開発プラットホーム

MDP01PCI

• 大規模プラットホームFPGAを中心に，TxDAC，RxADCなど，アナログフロントエンド回路を搭載．
• I/Qインターフェースは，シングルエンドと差動のどちらも利用可能．
• DSPが接続され，ソフト/ハードの協調実装が可能．
• 汎用D-AコンバータによりAGC制御が可能．
• PCIバスインターフェースを持つため，PCに搭載し，PC側で信号の収集や分析，あるいは通信プロトコルを組み込んで，高度な評価システムの構築が可能．
• ベースバンドアナログインターフェースを板間コネクタに配置しているので，RFモジュールを搭載可能．
• VC-TCXO（1ppm）を搭載し，FPGAのグローバルクロックに入力．
• 拡張用インターフェースを利用し，ドータボードを搭載可能で，D-AコンバータやA-Dコンバータを増設できる．

●価格：下記へ問い合わせ

■ （株）キューウエーブ
TEL/FAX：03-3485-2900

●リアルタイムエンコーダボード

DM-PC500

• パソコンのPCI拡張スロットに装着し，パソコンによって制御可能な，HDTV信号をMPEG-2信号にリアルタイムで圧縮するハードウェアエンコーダボード．
• SMPTE292M準拠のHD SDI信号入力を，MPEG-2ビデオ規格とMPEG-1オーディオ規格に準拠して符号化し，MPEG-2システム規格に準拠したトランスポートストリームとしてDVB/ASI規格により出力する機能をもつ．
• エンコーダ部分は，現在ディジタル放送局で使用されている送出用エンコーダとほぼ同程度の仕様と能力をもつ．
• 独自開発のMPEG-2エンコードアルゴリズムの採用により，高画質なエンコードが可能．

●価格：オープン価格

■ 日本ビクター（株）
TEL：0426-60-7292

●Linux対応FPGAボード

SUZAKU

• Xilinx社製のFPGA「Spartanシリーズ」をベースとしたボードコンピュータ．
• Xilinx社から提供されるソフトプロセッサ「MicroBlaze」に任意の周辺回路を加え，オリジナルのシステムLSIを設計し，組み込むことが可能．
• 72×47mmの小型サイズ．
• 100Base-TXのネットワークに対応．
• OSとしてLinux（μCLinux Kernel 2.4.22ベース）を採用することで，豊富なソフトウェア資産と安定性を提供．
• 機能のほとんどをFPGAの内部に実装しているため，次世代のFPGAに交代しても同一機能を実現できる．
• ボードの長期供給が可能．

●予定価格：¥25,000

■ （株）アットマークテクノ
TEL：011-890-6551
E-mail：info@atmark-techno.com
URL：http://www.atmark-techno.com/

●FPGA用エミュレータ

PALMiCE FPGA

- ターゲットシステムに搭載されたFPGAのJTAGポートと，専用コネクタで接続することで，動作中のFPGA内部のノードやブロックRAMの観察を可能にした実機デバッグツール．
- あらかじめI/O端子を割り付けておくことで，最大16チャネルのステートアナライザ機能の使用が可能．
- FPGA内部リソースを占有しない．
- 対応言語は，VHDLおよびVerilog-HDL．
- 対応FPGAは，Xilinx製Spartan-II/II E，Virtex，Virtex-E/II．
- ホストパソコンとは，USB（Ver1.1）による接続が可能．
- 操作性に優れたGUIをサポート．
- 「XI-FPGA380」とステートアナライザ機能を持たないベーシックモデル「XI-FPGA200」の2種類を用意．

●価格：¥398,000（XI-FPGA380）
¥298,000（XI-FPGA200）

●2004年3月末までのキャンペーン価格：
¥198,000（XI-FPGA380）
¥ 98,000（XI-FPGA200）

■（株）コンピューテックス
TEL：03-3253-2901　FAX：03-3293-2902
E-mail：sales@computex.co.jp

●Linux機器向け開発プラットホーム

STORM プラットホーム

- 64ビットMIPSプロセッサとLinuxをOSに使用した組み込み機器のハードウェア/ソフトウェア開発にかかる工数，コスト，時間を短縮することが可能な開発プラットホーム．
- PCM-Sierra社の低諸費電力MIPSベースプロセッサ「RM7065C-600C」を搭載しており，600MHzの高速プロセッシングが可能．
- システム制御チップとしてALTERA社の「Stratixシリーズ」FPGAを使用し，MIPSアーキテクチャ標準バスであるSysADバスとPCIバスのブリッジ機能，メモリ制御機能などを集積．
- FPGA内のSysADバス，PCIバス，SDRAMコントローラなどの機能ブロックには，Eureka社が開発したIPコアを含む．
- ユーザーロジック用にALTERA社「Cycloneシリーズ」FPGAを使用．
- 付属のEthernetポートをクロス開発環境が構築されたPCに接続するだけで，デバッグが可能．

●価格：¥198,000

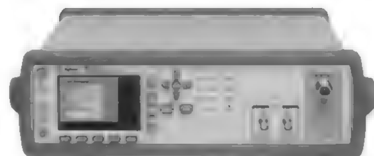
■（株）アルティマ
TEL：045-476-2045　FAX：045-476-2046

●Bluetooth/無線LANテスタ

N4010A ワイヤレス・コネクティビティ・テストセット

- RF測定や接続性試験など，Bluetoothデバイスや無線LANデバイスの機能と性能が検証可能なワンボックステスタ．
- 測定シーケンスの簡略化および自動化を実現する内部シーケンサを搭載したことで，従来品と比較して，主要な測定項目で30～50%の測定速度の向上が可能．
- ±0.5dBの高確度を実現することにより，製造時に必要な動作確認のための測定時間の短縮，スループットの向上を実現．
- 他のスペクトラムアナライザや信号源との接続端子を装備．
- 無線LAN測定のためには，オプションの追加が必要．

●価格：¥200,000,000～



■アジレント・テクノロジー（株）
TEL：0120-421-345

●ビジュアルコミュニケーションツール

Viewer Port

- FOMA「F2402」などの3G-324M規格に対応した通信カードとの組み合わせにより，テレビ電話対応のFOMA音声端末との間でリアルタイム音声，映像通信を行うことが可能．
- 映像，音声入力ポートを搭載することで，暗視カメラやドーム型カメラなど，設置場所に合わせたカメラ装置を利用することが可能．
- 家庭内のテレビ，ビデオカメラとの接続も行える．
- 各種センサを接続するための専用ポートを搭載しているため，接続したセンサに反応があった場合，指定した連絡先に自動的にテレビ電話の発信を行うことが可能．

●価格：下記へ問い合わせ



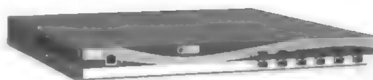
■（株）ハギワラシスコム
TEL：03-3517-1531　FAX：03-3517-6336

●SANファブリックスイッチ

Sphereon 4300

- SANを初めて導入する場合や小規模な部所での導入，エンタープライズ環境のEdgeなどをターゲットとした12ポートのエントリレベルのファブリックスイッチ．
- オンラインのまま中断なしでファームウェアをロードしてアクティベートできるHotCAT機能を備えている．
- 4ポートから導入可能で，FlexPortと呼ぶ機能により，8ポートもしくは12ポートまで中断なしで拡張可能．
- SAN管理は，付属のブラウザベースのSANpilot管理ソフトウェアを使用して行うことができる．

●価格：下記へ問い合わせ



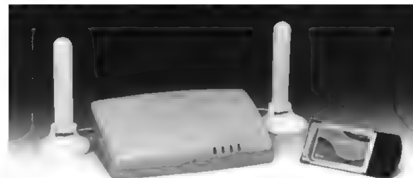
■マクデータ・ジャパン（株）
TEL：03-3512-3671　FAX：03-3512-3672

●無線LANシステム

WA-7000

- 2.4GHz/5GHz（4.9GHz～5.25GHz）のデュアルバンド対応で，IEEE802.11bおよびIEEE802.11a/gを同時に使用可能．
- IEEE802.11a/gでは，同時使用の場合も54Mbpsの高速アクセスを実現．
- クライアント認証機能を搭載し，暗号化技術をサポート．
- クライアント認証機能はIEEE802.1x：EAP-TLS，EAP-TTLS，PEAPに対応し，暗号化技術はダイナミックWEP，WPA1（TKIP），WPA2（AES）に対応．
- アクセスポイントの設定は独立したVLANからのみ可能となっており，クライアント端末からのネットワークの隠蔽などの不正操作を防止．

●価格：オープン価格



■（株）東芝
TEL：03-3457-2977

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

NEWPRODUCTS

●開発用プラットホーム

FR-Vソリューション・パッケージ

- 「FR-Vプロセッサ」を中核として，応用分野ごとにソフトウェアとハードウェアをパッケージ化した開発用プラットホーム．
- ソフトウェアパッケージ「Si Viewer Solution Software for PDK」および評価キット「PDK」で構成される．
- モバイル機器やテレビなどの機器と接続可能なインターフェースを装備しており，入力される画像や音声をリアルタイムに圧縮して記録し，記録した情報の再生などが可能．
- 画像や音声の圧縮形式など各種フォーマットの追加や変更は，ハードウェアに変更を加えることなく，ソフトウェアの追加や変更だけで対応できる．
- リファレンスアプリケーションをカスタマイズすることで，専用のアプリケーションとして利用可能．
- OSとして，アックスが提供するaxLinuxソフトウェア開発キット「axLinuxSDK 3.6」を別途購入する必要がある

●価格：¥500,000（シリコン・ビューワSPD）

■富士通（株）
TEL：03-5322-3354
E-mail：edevice@fujitsu.com

●数式処理/数値計算ソフトウェア

Maple 9 日本語版

- カナダのMaplesoft社が開発した，代数計算や微分方程式など広範な数学関数を備える数式処理，数値計算，グラフィックスプログラミングを統合したシステム．
- JavaやFortran，C言語などへのコード変換機能に加えて，制御，通信，信号処理分野で利用されている数値計算ソフト「MATLAB」や「Visual Basic」へのコード自動変換をサポート．
- 線形代数や初等微積分から，常微分方程式の解析まで利用できるGUIアプリケーションを提供する「Maplets」チュータ群により，Mapleの関数やコマンドを覚えることなく，直感的に計算や解析を行うことが可能．

●価格：¥241,000



■サイバネットシステム（株）
TEL：03-5978-2481　FAX：03-5978-6082
E-mail：infomaple@cybernet.co.jp
URL：http://www.cybernet.co.jp/maple/

●ITRON OS

TOPPERS/FDL-Pro

- Windowsサーバ上のSH-C，ADSなどの商用コンパイラでコンパイルしたITRON用のソフトウェアをローダブルモジュールとして，ターゲット機器へのダイナミックローディングを可能にしたITRON OS．
- プロトタイピングの段階では，ハードウェア完成前でも付属のシミュレータによりローダブルアプリケーションの個別先行開発が可能．
- ハードウェア完成後は，LAN環境に接続することで，複数のエンジニアによるアジャイルな開発が可能．
- 製品出荷後は，不具合対策や新機能追加を通信インフラによって実現．
- ITRON資産を活かしつつ，従来型のITRON仕様OSやオープンソースソフトウェアでは得られない生産性，信頼性，メンテナンサビリティを得ることが可能．

●価格：下記へ問い合わせ

■（株）エーアイコーポレーション
TEL：03-3493-7981　FAX：03-3493-7993
E-mail：sales@aicp.co.jp
URL：http://www.aicp.co.jp/

●Java開発プラットホーム

Borland JBuilder X 日本語版 Borland Optimizeit Suite 6.0 for Java 日本語版

- コード入力支援機能に加え，コンパイル前に構文エラーを検出する「ErrorInsight」の強化，各種フィルタリング機能や変数名の一括変換が可能な機能など，コーディング作業の効率化を実現．
- 複雑なStrutsを使ったWebアプリケーションの開発に，ドラッグ&ドロップによるビジュアル操作とコードによる開発を両立させる2Wayテクノロジを導入．
- WSDL，UDDIによるWebサービスの活用，Javaクラス，EJBなどのWebサービスによる公開を実現するWebサービスデザイナを搭載．
- 「Borland Optimizeit Suite 6.0 for Java」は，コードに関する単純なレベルの問題から，J2EEコンポーネントに関係するボトルネックに至るまで，さまざまな問題の解決に必要な機能を開発者に提供．

●価格：Enterprise版 ¥300,000
Developer版 ¥68,000
Foundation版 無償ダウンロード
Optimizeit Suite 6.0 for Java ¥190,000

■ボーランド（株）
TEL：03-5323-3071　FAX：03-5323-3072

●.NET開発環境

Boland Together Edition for Microsoft Visual Studio .NET

- Visual Studioの開発環境にシームレスに統合され，先進のモデリング機能の利用が可能．
- 「Borland LiveSource」によって，つねにモデル図とアプリケーションの同期が保証されるため，ツール間でのインポートやエクスポート，バッチ処理は不要．
- クラス，シーケンス，コラボレーション，アクティビティ，状態，コンポーネントなど，頻繁に必要とされる図や表記をサポート．
- 各ダイヤグラム間の図上の要件，外部のドキュメントやURLとダイヤグラムとのハイパーリンクを可能とし，他のモデリングツールとのインポート/エクスポートを可能にするXML形式をサポート．
- 標準的なコーディングパターンを再利用するためのシンプルコード形式のパターンをサポートし，シンプルなソースコード形式ファイルによる拡張可能なテンプレートベースのパターンやGoFパターン，C#，その他の共通パターンを含む事前定義パターンの再利用も可能．

●価格：¥98,000（指名ユーザーライセンス）
¥190,000（フローティングライセンス）

■ボーランド（株）
TEL：03-5323-3071　FAX：03-5323-3072

●日本語入力ソフトウェア

MobileWnn IMF for Qt/Embedded

- Linuxのデスクトップ環境などで幅広く利用されている，Wnn入力システムの組み込み用ソフトウェア．
- MontaVista Linux Professional Editionをサポート．
- 携帯電話，PDA，ゲーム機器などのメモリサイズが制限されたコンシューマエレクトロニクス製品で，日本語入力GUI環境の実装を可能にする．
- 入力予測，連文節変換，学習機能，オプション辞書などの機能をもつ．
- 単語登録，環境設定GUIを提供．
- ソフトウェアキーボードによる入力をサポート．
- システム，用途に応じたカスタマイズが可能．

●価格：下記へ問い合わせ

■オムロン ソフトウェア（株）
TEL：044-246-6016　FAX：044-246-6011

●組み込み向けかな漢字変換ソフトウェア

Compact-VJE

- バックス社が開発した，組み込み用途向けかな漢字変換ソフトウェアを，神戸製鋼が各種組み込みOSに移植した製品.
- 組み込み用途に適したコンパクトサイズを実現.
- 連文節/単文節/単語/単漢字変換の各バージョンを用意.
- 3万語辞書が標準で付属し，オプションで5万/6万/8万/23万語辞書を用意.
- 辞書メンテナンスツールによる辞書のカスタマイズが可能.
- 学習機能対応可能.
- ソースコードを提供.
- 動作確認済み対応OSは，μiTRON v4（HI7700，NORTi）および，Tornado/VxWorks.
- 動作確認済み対応CPUは，μiTRON対応版SH-4（SH7750/SH7751）および，Tornado/VxWORKS対応版Pentium.

●価格：下記へ問い合わせ

■ 東電ユークエスト（株）
TEL：03-6402-2051 FAX：03-3432-1061
E-mail：sales@uquest.jp
URL：http://www.uquest.jp/

●組み込み向けTCP/IPプロトコルスタック

PrCONNECT2

- TCP/IPプロトコル使用時の通信で，実測値約50Mbps.
- ドライバとプロトコルスタック部分の切り分けが明確となり，容易なドライバ構築を実現.
- TMS320DM270用Ethernetドライバをサンプルで提供.
- PPPなど各種プロトコルを標準提供.
- UNIX BSDソケットのサブセットを提供.
- ITRON TCP/IP API仕様準拠のAPIを提供.
- システム編集スイッチを採用することで，環境に合ったサイズにコードの縮小が可能.
- Ethernetドライバ，PPPドライバ，FTPアプリケーション，TFTPアプリケーションなど各種サンプルを標準提供.

●価格：下記へ問い合わせ

■ イーソル（株）
TEL：03-5302-1360 FAX：03-5302-1361
E-mail：ep-inq@esol.co.jp
URL：http://www.esol.co.jp/embedded/

●インストレーションオーサリングソリューション

InstallShield DevStudio 9 日本語版

- Windows Installer（MSI）やInstallScript，smart device formatに準拠したインストレーションソリューションを実現.
- MSIやInstallScriptなど，複数のインストレーションプロジェクトのタイプから適切なものを選択することが可能.
- Windows CEや携帯電話などに対するプロジェクトタイプをサポート.
- すべてのプロジェクトタイプに共通の開発環境を提供し，Visual Studio .NET 2003環境下での動作が可能.
- 従来製品のプロジェクトを，シームレスに移行させることが可能.
- 直感的なウィザードやアシスタントを採用し，複雑なセットアッププロジェクトを容易に作成可能.
- 従来製品と共存でき，テスト過程や開発の簡素化を実現.

●価格：¥228,000

■（株）ネットワールド
TEL：03-5210-5187 FAX：03-5210-3912

●日英/英日双方向翻訳ソフトウェア

PC-Transer V11 for Windows

- 日英翻訳エンジンの改良を行い，カタカナのゆらぎに対応.
- レイアウトや文字スタイルなどの保持が可能な，WORD/EXCELオフィスアドイン翻訳，辞書ごとにカラー表示が可能で，大量文書を上下に分割して翻訳可能な対訳エディタ，ワンタッチ辞書引き「ロボワード for トランサー」，英語の認識率が高い文字認識ソフトウェア「OmCR」などを搭載.
- 文書に合った専門辞書を自動で選択するオート専門語辞書セレクト機能を搭載し，ビジネス版で7分野64万語，科学技術版で16分野139万語，総合版で25分野233万語の専門辞書を搭載.
- 翻訳メモリエンジンを搭載することで，100万例の翻訳メモリでも，ストレスを感じさせないスピードで検索して訳文を生成.
- 翻訳辞書と文法ルールを使って構文解析を行う機械翻訳と，文単位での検索および訳文生成を行う翻訳メモリを統合することで，実用的な翻訳業務支援を実現.

●価格：¥48,000（ビジネス版）

■（株）クロスランゲージ
TEL：03-5287-7588
E-mail：info@transer.com

●CAD設計データ管理/部品構成管理システム

AutoServer4 BOM Data Server

- AutoServer4は2次元/3次元CADの設計図面データ/設計情報管理システム，BOM Data Serverは2次元の部品表，3次元のアセンブリ情報などの部品構成管理システム，Engineering Information ServerはAutoServer4とBOM Data Serverおよび導入支援コンサルティングをワンパッケージ化.
- AutoServer4は，図面と技術文書を同じように扱う機能を装備し，AutoCADセキュリティ機能に対応．AutoMECH2004/AutoMECH LT2004のハンコ機能と連動する承認フローを実装.
- BOM Data Serverは，部品構成管理に特化した専用サーバで，部品表/部品表項目/パーツ図面など部品構成管理に必要な機能を装備．Webサービスにより他のシステムとの連携が可能で，DCOMによるインターフェースを提供．マイクロソフト社の.NET Framework 1.1に対応.

●価格：
¥1,200,000～（AutoSever4）
¥1,800,000～（BOM Data Server）
¥4,000,000～（Engineering Information Server）

■（株）エス．アール．ディー
TEL：06-6392-9511 FAX：06-6392-9524

●入力インターフェースコンポーネント

InputMan for .NET 2.0

- アプリケーションの基本となる入力インターフェースを支援する，テキスト，マスク，日付け，数値，コンボ，カレンダ，電卓，コンテナ，ファンクションキーの九つのコントロールで構成されたWindowsフォーム用コンポーネント.
- コンボコントロールは，マスク書式を設定可能なテキストボックス部分と画像，項目，説明文の三つの列を表示するリストボックス部分で構成される.
- 初期状態に戻せるソート機能やオブジェクト型にも対応した検索機能，項目ごとのツールチップ，リストボックス最下部のステータスバーなどの機能を搭載.
- .NET版として開発されたファンクションキーコントロールは，キーフック機能と画像表示やグループ化などの豊富なカスタマイズ機能を備えたボタンの活用により，複数のウィンドウを切り替えることの多いシステムで，円滑な入力を支援する.

●価格：下記へ問い合わせ

■ グレープシティ（株）
TEL：022-777-8211 FAX：022-777-8233
E-mail：sales@grapecity.com

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）

IPパケットの隙間から

知らないうちに危険なことをしている人々

祐安 重夫

筆者がLinuxを使ったデータベースサーバを管理している会社から電話があり，サーバにつながらなくなったという．話をよく聞いたところ，プロバイダを変更したのだという．つながらなくなるのは当然だ．プロバイダを変更するなら，あらかじめ連絡してもらわないと困ってしまう．まして，相手のサーバが設置してあるのは大阪である．

岐阜に設置してある別の組織のサーバの場合は，去年のはじめにプロバイダの変更があり，この時はあらかじめ予定の連絡があってからスケジュールを調整して見積を出し，出かけていった．基本的には，このような手順を踏んでもらわないと，都内ではないのだから急に対応できるわけはない．

今回の依頼元は，どうやらプロバイダが変更になるとIPアドレスなどが変わるという基本的なことさえ，認識していなかったようなのだ．Windowsクライアントもいくつかあるはずなので，それはどう対応したのかと思ったら，すべて手動でネットワークの設定を変更したらしい．とりあえず新しいプロバイダからのIPアドレスなどの情報と，現在クライアントに割り当てていないIPアドレスを連絡してもらった．

しかたなしにLinuxにrootでログインしてネットワーク情報を変更する方法と，ファイアウォールで通過させる必要があるポートについてメモを作り，それを元にして先方に作業してもらうことにした．

ファイアウォールは，前のプロバイダのときからすでに設置してあったので，それがどうなっているかがちょっと気にはなったが，すでにクライアントマシンは接続できているようだし，ファイアウォールでもLAN側にローカルアドレスではなくグローバルアドレスを使用する設定が可能なものもある．それに，ファイアウォールの設定は，最初にサーバを設置した時点から大阪の業者が行っており，筆者の仕事の範囲外となっている．

その後，しばらく連絡がなかったので，無事に動作しているのだろうと思っていたら，1週間ほどして電話がかかってきて，やはり動かないので何とか大阪まで行ってきてほしいという．ともかく現状がどうなっているのか，東京の担当者と話をしても，ただ動かないとしかわからない．ようやく大阪の担当者からのメールが転送されてきて，Linuxサーバの設定は指定どおりにできたようだが，ファイアウォールについてはもともと外注したので，設定できないという．

ここまできて，どうやら先方は，プロバイダを変更しても一度設定したファイアウォールは，そのまま使用できると思っていたらしいことがわかった．それではクライアントマシンはどうして動作しているのかと思ったら，ファイアウォールの内側に接続してもWebの閲覧もメールの送受信もできないため，外側につないだのだという．その際に，クライアントマシンにグローバルアドレスを設定してしまったらしい．かなり危険な状態だと思うが，Windowsクライアントについては筆者の関知するところではない．

そういえば数年前に最初にサーバを準備したときも，東京でOSのインストールから必要な設定まで済ませ，ネットワークについてはその時点で予定されていたグローバルアドレスにしておいた．ファイアウォールについては，実際に大阪に設置される際に現地の担当業者から連絡があり，一度ファイアウォールの外につないでもらい，リモートで接続してローカルアドレスへの設定変更を行い，ファイアウォールの内側に接続し直してもらって，もう一度リモート接続ができることを確認した．それからつい先日までは，ずっとリモートメンテナンスを行ってきたのである．

ということは，すでに設定が終了しているはずのLinuxサーバを，ファイアウォールの外に直結すれば動作するはずだとも思えるが，そんな危険なことをするわけにはいかない．

半年ほど前にOSのバージョンアップを行いたいので一時的にサーバを東京に送り返すか，大阪までの出張費を出すかを選択してほしいと連絡したことがある．OSのバージョンアップ自体はサポート契約の範囲内だが，出張が必要な場合は交通費と宿泊費は別途精算するという約束だったのだ．しかし今の時点では余分なお金はかけたくないという返事が来て，そのまま保留になっていた．そして数か月前には，サポート契約の更新はしないということになったのだった．

もし，OSをバージョンアップしていれば，Linuxに組み込まれたファイアウォール機能が使用できたのだが，現状ではファイアウォールの中にあるということで，古いバージョンでも問題はないだろうということになったのが，こんなところで仇になった．とはいっても，バージョンアップしないことを選択したのは先方である．

しかし，これまでのやりとりで判明したことは，先方はプロバイダの変更の際にファイアウォールを設置した業者と何の連絡もとっておらず──というより連絡が必要なことにさえ気がついておらず──，そのままで動作すると思い込んでいたらしいということである．

ふと気がついたのだが，今回は，じつはファイアウォールの設定変更だけを行えば，LAN上のクライアントもサーバも何の変更もなしにそのまま動作し続けたのではないだろうか．

すけやす・しげお　インターメディアアクセス


INFORMATION
海外・国内イベント/セミナー情報


海外イベント

2004/1/5-9 **International Conference on VLSI Design**
Renaissance Hotel, Mumbai, India
International Conference & Exhibition Services
http://vlsi.nj.nec.com/

1/5-9 **Macworld Conference & Expo**
The Moscone Center, San Francisco, CA, USA
IDG
http://www.macworldexpo.com/

1/8-11 **2004 International CES**
Las Vegas Convention Center / Las Vegas Hilton / Alexis Park, Las Vegas, NV, USA
Consumer Electronics Association
http://www.cesweb.org/

1/16-18 **Wi-Fi Home & Small Office Expo**
Jacob K. Javits Convention Center, NY, USA
Jupitermedia
http://www.jupiterevents.com/

1/20-23 **LinuxWorld Conference & Expo**
Jacob K. Javits Convention Center, NY, USA
IDG
http://www.linuxworldexpo.com/

1/25-29 **17th IEEE International Conference on Micro Electro Mechanical Systems 2004**
Maastricht Exhibition and Convention Center, Maastricht, Netherland
IEEE
http://www.mems2004.org/

1/26-29 **COMNET Conference & Expo**
Washington Convention Center, Washington D.C., USA
IDG
http://www.comnetexpo.com/

2/14-19 **ISCC(IEEE International Solid-State Circuits Conference)**
San Francisco Marriott Hotel, San Francisco, CA, USA
IEEE Solid-State Circuits Society
http://www.isscc.org/isscc/

国内イベント

2004/1/27-30 ASP-DAC 2004
パシフィコ横浜（神奈川県横浜市）
日本エレクトロニクスショー協会
http://www.aspdac.com/jp/index.html

1/28-30 第33回インターネプコン・ジャパン/第21回エレクトロテスト・ジャパン/第5回プリント配線板EXPO/第5回電子コンポーネントEXPO/第5回　半導体パッケージング技術展/第4回ファイバーオプティクスEXPO
東京国際展示場（東京ビッグサイト，東京都江東区）
リードエグジビションジャパン
http://www.reedexpo.co.jp/inj/

1/29-30 Electronic Design and Solution Fair 2004
パシフィコ横浜（神奈川県横浜市）
日本エレクトロニクスショー協会
http://www.edsfair.com/

2/4-6 NET&COM 2004
日本コンベンションセンター（幕張メッセ，千葉県千葉市）
日経BP社
http://expo.nikkeibp.co.jp/netcom/

2/4-6 ISS（SEMI Industry Strategy Symposium）Japan 2004
パンパシフィックホテル横浜（神奈川県横浜市）
社団法人溶接学会
http://www.semi.org/wps/portal

3/2-5 IC CARD WORLD 2004
東京国際展示場（東京ビッグサイト，東京都江東区）
日本経済新聞社
http://www.shopbiz.jp/pages/t_index.phtml?PID=0003&TCD=IC

セミナー情報

ビギナのためのアナログ回路設計
開催日時　：1月10日（土）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：3,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

電磁界シミュレータでわかる高周波技術
開催日時　：1月15日（木）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

SystemCを正しく理解する為のC++入門講座
開催日時　：1月16日（金）※毎月1回開催
開催場所　：川崎産業振興会館（神奈川県川崎市）
または BIZ 新宿（東京都新宿区）
受講料　：9,800円
問い合わせ先：（株）礎デザインオートメーション営業部，☎03)6762-1472，FAX(03)6762-1472
http://www.ishizue-da.co.jp/products/seminar_c++.htm

OPアンプの基礎と応用
開催日時　：1月17日（土）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

C/SystemCからの標準的RTL変換手法
開催日時　：1月21日（水）～1月23日（金）※毎月1回開催
開催場所　：川崎産業振興会館（神奈川県川崎市）
または BIZ 新宿（東京都新宿区）
受講料　：49,800円（1日1部，全3部構成．1月21日は無料）
問い合わせ先：（株）礎デザインオートメーション営業部，☎(03)6762-1472，FAX(03)6762-1472
http://www.ishizue-da.co.jp/products/seminar_c2rtl.htm

XPortソリューションセミナー
開催日時　：1月21日（水），1月23日（金）
開催場所　：日新システムズ京都本社（1月21日開催，京都府京都市下京区），日新電機東京支社（1月23日開催，東京都千代田区）
受講料　：55,000円（「X-Port Evaluation Kits」1セット付き）
問い合わせ先：（株）日新システムズ，<関西地区>☎(075)344-7881，FAX(075)344-7887，<関東地区>☎(03)5807-5931，FAX(03)3839-0112
http://www.co-nss.co.jp/event/event.html#xport

USBの基礎と応用
開催日時　：1月22日（木）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

PLL回路の設計法
開催日時　：1月24日（土）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

Linux組込み設計手法～組込み向けROM化Linux"Silicon Linux"とSHボード～
開催日時　：1月29日（木）
開催場所　：中央大学駿河台記念館（東京都千代田区）
受講料　：62,800円
問い合わせ先：（株）トリケップス，☎03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/040129a.htm

Linuxプロフェッショナル協会認定LPIC取得者によるLinux技術者認定試験の概要と資格取得のポイント
開催日時　：1月29日（木）～1月30日（金）
開催場所　：SRCセミナールーム（東京都高田馬場）
受講料　：76,000円
問い合わせ先：（株）ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/24/24_019.htm

TCP/IPソケットプログラミング入門
開催日時　：1月30日（金）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

デバッグ工学　組み込みソフトウェアのシステム設計とテスト技法
開催日時　：2月2日（月）～2月3日（火）
開催場所　：SRCセミナールーム（東京都高田馬場）
受講料　：76,000円
問い合わせ先：（株）ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/24/24_023.htm

ソフトウェア開発における要求の仕様化と管理法
開催日時　：2月2日（月）
開催場所　：オームビル（東京都千代田区）
受講料　：52,500円/1口（1口で1社3名まで受講可）
問い合わせ先：（株）トリケップス，☎03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c040202n.htm

オブジェクト指向技術によるプロセス改善の実践
開催日時　：2月5日（木）
開催場所　：CQ出版セミナールーム（東京都豊島区）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

開催日，イベント名，開催地，問い合わせ先の順

日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

読者の広場

Interfaceへの声



2003年12月号特集「具体例で学ぶ組み込みソフトの再利用技術」に関して

▷今月の特集は非常に読み応えがありました．具体例もイメージしやすいもので良かったです．今後の仕事にぜひ活用したいと思います．ただしベースのポットに本当にRTOSを採用するかは疑問ですね．
(moto)

［編］もっともベーシックな電子ポット単体を考えると，たしかにRTOSは不要だと思われるでしょう．特集ではその後の商品シリーズ展開も考慮した設計事例を解説しています．このあたりは，たしかに判断の難しい部分だと思います．

▷仕様をみて機能を実現するために，どのように実装するかを追及して開発してきました．いわゆる「改良」もひと段落して，いままで築いてきたはずの要素技術をどう整理して今後“楽”をするか悩んでいましたが，ちょっと方針らしきものがつかめました．これからの「改良」に役立てたいと思います．
(常盤 稔)

▷私も組み込み系システムの開発を行っている．大手メーカーのプロジェクトなのだが，UMLなどの設計技法はほとんど使われていない．以前いたメーカーでも通信システムなのに簡単なシーケンス図も見せてもらえず，どうやってプログラムを組むのかさっぱりわからなかった記憶がある．これからもどんどん再利用技術の啓蒙を行ってください．
(独身チョンガーズ)

Interface全般に関して

▷特別付録の「エンジニアの技術草子」がよかったです．1年に2回くらいこのようなコラムを集めた特別付録があれば，Interface誌のちがう側面が見られて，また国外の雑誌にない味があってうれしいと思うのは私だけでしょうか？
(JR9JUK)

［編］「エンジニアの技術草子」は人気の高いコラムの一つです．筆者の旭氏には，今後もさまざまな視点で，技術者のためのコラムを執筆していただく予定です．ご期待ください．

▷InterGigaを付けるのは良いが，展示会などの特集が少なくなってきている．もっと増やしてほしい．次号のPCI & PCI-Xの特集を楽しみにしています．
(てろてろ)

アンケートの結果

興味のあった記事（2003年12月号で実施）

①第1章 組み込みシステムの特徴を考える
②プロローグ 魅力ある製品を生み出すためのソフトウェア開発とは？
③第2章 電子ポット商品群にプロダクトラインを適用してみる
④Appendix 1 カラーで見るUML図（ユースケース図，クラス図，コラボレーション図）
⑤第3章 コア資産を摘出するためのドメインエンジニアリングの実際
⑥第4章 電子ポット商品群のコア資産をC++で実装する
⑦第5章 組み込みシステムのテスト手法
⑧エピローグ 組み込みソフトウェアエンジ

特集担当デスクから

☆C++のテンプレート機能――C++の教科書にも登場することから，名前自体は耳にしたことのある読者も多いだろう．そしてその代表的な使用例であるSTL，こちらも日本語の書籍もいくつか発行され，有名になってきた．
☆しかし，実際のプロジェクトでこれらを使っている例はまだまだ少ないようだ．「便利なことはわかっているけれど…」という言葉の裏には，使い方が難しい，多人数のプロジェクトで使用するためには，全員が使い方を熟知していなければならないなど，導入への障壁が高いことが伺える．そこでこの特集を理解して，実際のプロジェクトで使って欲しい．
☆また，最近では第2のテンプレートライブラリ，Boostが注目されている．STL++的な側面をもち，STLを使ったことのある人にはなじみやすい．また，マルチプラットホームで動作するファイルアクセスライブラリにより，Windowsと各種UNIXの間で同一のソースを共有する時の障壁となるファイル名の問題の回避など，便利な機能も多い．残念ながら現バージョンではこの部分に問題があるようだが，今後に期待したいところだ．
☆また，C++はちょっと…という方のために，C言語で使えるコンテナライブラリの解説も掲載した．リストや木構造などを毎回書き下ろしてはいないだろうか？　もしくは内製であまり実績のないライブラリを使ってはいないだろうか？　すでに安定したライブラリが存在するならば，それを使うに越したことはないだろう．
☆queue.hは，NetBSDのカーネル内部で使われるなど，「OSカーネルで使えるくらい」の安定性と省資源性，性能を誇る．また，glibで提供される数々の機能も便利に違いない．どちらも枯れたライブラリであるうえに，ライセンスもBSD/LGPLであり，組み込みにもうってつけだ．
☆明日からといわず，今日から使って欲しい．



ニアとコミュニティ
⑨別冊付録 シニアエンジニアの技術草子——特別編
⑩XScaleプロセッサ徹底活用研究(第4回)
⑪高性能圧縮ツールbsrcの理論と実装(前編)
⑫やり直しのための信号数学(第19回)
⑬TOPPERSで学ぶRTOS技術(第3回)
⑭初級ドライバ開発者のためのWindowsデバイスドライバ開発テクニック(第3回)
⑮「IrFront H8S Trial Kit」の概要
⑯第5回 自動認識総合展
⑰音楽配信技術の最新動向(第7回・最終回)
⑱ハッカーの常識的見聞録(第36回)
⑲Engineering Life in Silicon Valley(対談編)
⑳プログラミングの要(第8回)
㉑開発技術者のためのアセンブラ入門(第22回)

『具体例で学ぶ組み込みソフトの再利用技術』についてのアンケートの結果

Q1 今月号の特集はいかがでしたか?
①よく理解できた(27%)
②ある程度理解できた(64%)
③あまり理解できなかった(9%)
④ほとんど理解できなかった(0%)
⑤関係ない分野だった(0%)
⑥その他(0%)

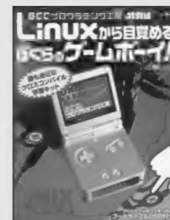
Q2 特集のキーワードになっている「体系的な再利用」について,ご自分の開発との関係は? またその理由は?
①手がけている開発に採用できそう(55%)
理由:組み込みではなくPCアプリケーション開発でも同様の考え方はできる
②手がけている開発には使えなさそう(18%)
③その他(27%)
理由:参加している人数が多いプロジェクトなので,自分一人だけではなんともいえない

Q3 組み込みシステムに関して,どんな記事を期待されますか?
- RTOS採用の可否.とくにCPUのコスト(ROM/RAM容量)とソフトウェア開発のトレードオフに関して.RTOSが本当に必要かどうか
- 開発環境構築方法
- たとえばZ80→H8などのCPU変更に際したシステム移植テクニックなど
- 熱解析の方法と精度について


読者プレゼント


●応募方法:本誌読者アンケートはがきに必要事項を記入のうえ,2004年1月31日(必着)までにご応募ください.なお当選者の発表は発送をもってかえさせていただきます.

(1) Linuxから目覚めるぼくらのゲームボーイ (1名)
西田 亙 著
ISBN 4-7973-2564-X
ソフトバンク パブリッシング(株)



(2) 未来ロボット技術研究センターTシャツ (1名)
(http://www.fuRo.org/)
サイズ:L



Interface 次号予告

2004年3月号は
1月24日発売です

間違いやすいコーディング例の紹介からLinuxのROM化まで

Cプログラミングの基礎知識

間違いやすい例/コーディングの違いと最適化/関数作成の勘所/デバッグの前準備/組み込みC/組み込みの実例

C言語は演算子の記号が特殊で，＝と＝＝，&と&&，¦と¦¦などのように間違えやすい演算子が少なくありません．これらの演算子は，日常的な数学（算数?）の常識やほかの言語と比較してかなり特殊なため，コーディングのミスを誘発しやすく，気付きにくいといった面ももっています．

そこで次回はわかりやすく読み解けるように，A君（20歳くらいの初級エンジニア），B君（経験5，6年の若いエンジニア），C君（組み込みエンジニア）とN先輩（経験豊かな先輩エンジニア）の4人が登場し，先輩エンジニアとの掛け合いで話が進みます．

まず第1章では，ANSI Cで間違いやすいコーディング例を紹介します．第2章では，コーディングの違いと最適化例を例題で示します．第3章は関数の作成にあたって指針を簡単に紹介していきます．第4章ではデバッガを使わない一般的なデバッグ論について解説します．第5章，第6章で組み込みCプログラミングの実例を紹介します．

編集後記

●デジカメ，薄型テレビ，DVDレコーダを称して「新三種の神器」というらしい．これらの商品でヒットした企業は好決算を出しています．ディジタル家電用の高付加価値商品に使われる半導体は日本勢の独壇場です．半導体業界の白石と黒石がひっくり返るオセロゲームのような展開に，明日の希望を見る思いです．（檀）

●ここ数か月，特集やら増刊やらの担当が続きましたが，ETも終わったところで，一息つきながら来年/来期の特集やら増刊やらの企画をいろいろと練っているところです．社内的にも大きな組織変更やら方針転換やらがありまして，根本から練り直さないとならない企画があったり…さて，どうしたもんか…．（M）

●長時間にわたって集中力を維持するのが難しい状態になっている．確かに作業効率を考えると，1時間ごとに休憩を入れて続けた方が良いということはわかっているのだが，それをするのも難しい．で，集中力が途切れてしまうと，間違いも多くなるし，効率も悪くなる．何とかせねばならんとは，十分にわかっているのだが，さて……．（＝IO）

●コピーワンスによるバックアップの制限，PCでのディジタルキャプチャ不可，B-CASカードによる個人情報掌握の可能性，アナログではぎりぎり越境受信できていた局が映らなくなる…無理して地上ディジタル放送を開始しなくても，すでに整っている光ファイバ網を活用したほうが安上がりだと思うのですが．（み）

●本日（2004年12月1日）よりテレビの地上ディジタル放送が開始．編集部のみんなが忙しく働いているなかカウントダウンイベントを見てしまった（←仕事しろ!）．2011年7月には現行のアナログ放送が終了予定．あと6年7か月以内にわが家のテレビとお別れする日がやって来るのかと思うと，悲しい…．（もみ）

●携帯電話の新しい機能に魅力を感じていても，買い替えるのが面倒なのでそのまま使っていたら，ついに壊れました．仕方がないので新しいのを買ったら，写メールはもちろんのこと（いままではメールだけだった），いろいろな機能が満載．でも，こんなに機能が付いていても使いこなせそうにありません…．（Y2）

●イラクで日本人の死亡者が出た．小泉首相は終始一貫して「状況を見極めてしかるべき判断をする」との発言を繰り返している．自衛隊派遣は今後の米国との関係を考えると止むを得ない選択だとしても，国民に対して現在のイラクの状況としかるべき判断の基準を明確に示す責任があるのではないか．（ちゃん）

●ある日，目が覚めると…な，なんと部屋の隅にあるはずのない扉が!さらに扉を開けると，奥にはあるはずのない広ーい部屋が!! わーい狭いお部屋が広くなった～．なんておバカな夢を実際に見るくらい引っ越し熱が高まっている今日この頃．本格的な引っ越しシーズン前に，はやく「私のお城」をゲットせねば!（な）

お知らせ

■ 読者の広場

本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

■ 投稿歓迎

本誌に投稿をご希望の方は，連絡先（自宅/勤務先）を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です（送り先はsupportinter@cqpub.co.jpまで）．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

■ 本誌掲載記事についてのご注意

本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．

本誌掲載記事をCQ出版（株）の承諾なしに，書籍，雑誌，Webといった媒体の形態を問わず，転載，複写することを禁じます．

■ コピーサービスのご案内

本誌バックナンバーの掲載記事については，在庫（原則として24か月分）のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

● コピー料金（税込み）
  1ページにつき100円
● 発送手数料（判型に関わらず）
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
● 送付金額の算出方法
  総ページ数×100円＋発送手数料
● 入金方法
  現金書留か郵便小為替による郵送
● 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
● 宛先
  〒170-8461 東京都豊島区巣鴨1-14-2
  CQ出版株式会社 コピーサービス係
  （TEL：03-5395-4211，FAX：03-5395-1642）

■ お問い合わせ先のご案内

● 在庫，バックナンバー，年間購読送付先変更に関して
  販売部：03-5395-2141
● 広告に関して
  広告部：03-5395-2133
● 雑誌本文に関して
  編集部：03-5395-2122
  記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface

©CQ出版（株） 2004 振替 00100-7-10665
2004年2月号 第30巻 第2号 通巻第320号
2004年2月1日発行（毎月1日発行）
定価は裏表紙に表示してあります

発行人／増田久喜
編集人／山形孝雄
編集／大野典宏 村上真紀 山口光樹 大竹友美 小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森 祐子
広告／澤辺 彰 中元正夫 菅原利江

発行所／CQ出版株式会社 〒170-8461 東京都豊島区巣鴨1-14-2
電話／編集部（03）5395-2122 FAX／（03）5395-2127
広告部（03）5395-2133 URL http://www.cqpub.co.jp/interface/
販売部（03）5395-2141 E-mail supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社 美和印刷株式会社
製本／星野製本株式会社


日本ABC協会加盟誌
（新聞雑誌部数公査機構） ISSN0387-9569